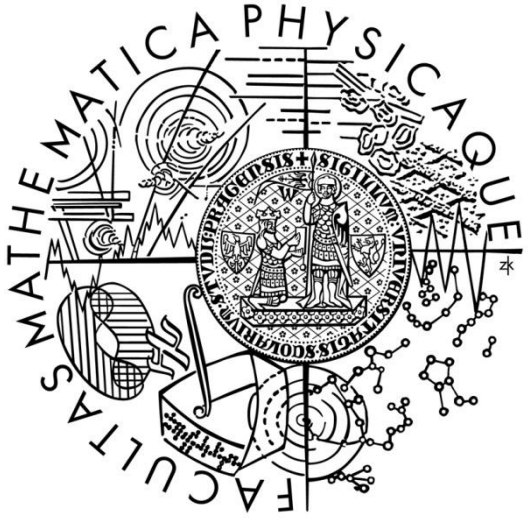
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jiří Korchňák

Advanced Interface for XML Data

Department of Software Engineering

Supervisor of the master thesis: RNDr. Irena Mlýnková, PhD.
Study programme: Informatics
Specialization: Software systems

Prague 2012

I would like to thank to my supervisor RNDr. Irena Mlýnková, Ph.D. for her helpful suggestions, thorough notes, provided related research material and text corrections.
I would also like to thank to my girlfriend Lucie Koutská, for her tolerance and comfortable words spoken at tough time.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date Jiří Korchňák

Table of contents

Abstract 8
1 Introduction 9
1.1 Thesis Organization 9
2 Existing XML APIs 10
2.1 DOM 10
2.1.1 Level 0 11
2.1.2 Intermediate DOM 11
2.1.3 Level 1 12
2.1.4 Level 2 13
2.1.5 Level 3 13
2.2 SAX 15
2.3 StAX 16
2.3.1 Cursor API 17
2.3.2 Event Iterator API 18
2.4 JAXP 18
2.5 .NET XML 19
2.6 Resume 21
3 Other Techniques 23
3.1 A DOM Method to Retrieve Data from a Very Large XML Document 23
3.1.1 Partitioning 23
3.1.2 Padding 25
3.1.3 Retrieving 27
3.2 Hybrid Parallelism for XML SAX Parsing 29
3.2.1 Stage One - Preliminary Parsing 30
3.2.2 Stage Two - Chunk Resolution 31
3.2.3 Stage Three - Namespace Processing 31
3.2.4 Stage Four – Callbacks 32
3.2.5 Execution 32
3.3 Lazy XML Parsing/Serialization based on Literal and DOM Hybrid Representation 34
3.3.1 Partial XML Parsing and Partial XML Tree Construction 34
3.3.2 Serialization Using the Original Literal XML Representation 35

3.3.3 Deferred Update Using the Change History 36
3.4 Prefiltering Techniques for Efficient XML Document Processing 37
3.4.1 Indexer 38
3.4.2 Query Simplifier 39
3.4.3 Fast Lightweight Steps-Axes Analyzer 39
3.4.4 Fragment Gatherer 40
3.4.5 Micro XML Streaming Parser 40
3.5 ORDPATHs: Insert-Friendly XML Node Labels 40
3.5.1 Concepts 41
3.5.2 L_i/O_i Pair Design 42
3.6 Resume 43
4 Proposed API 45
4.1 Motivation 45
4.2 Question-form Analysis 45
4.3 Simple DOM API for Large XML Files 47
4.3.1 API interface 47
4.3.2 Algorithm for Data Retrieving 48
4.3.3 Algorithm Split 48
4.3.4 Algorithm Retrieve 50
5 Implementation 52
5.1 Node Class 52
5.2 LargeXmlDocument Class 53
5.3 Element Class 54
5.4 Attribute Class 55
5.5 Config Class 56
5.6 XmlSplitter Class 56
6 Experiments 57
6.1 Testing Scenario A 57
6.1.1 Test Data 57
6.1.2 Test Results 59
6.2 Testing Scenario B 61
6.2.1 Test Data 61
6.2.2 Test Results 63
7 Conclusions 64

7.1 Future Work 64
Bibliography 65
Appendix A 70
Appendix B 71
Appendix C 72
Appendix D 76

Název práce: Pokročilé rozhraní pro XML data

Autor: Jiří Korchňák

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková, PhD.

e-mail vedoucího: irena.mlynkova@mff.cuni.cz

Abstrakt:

V současnosti existuje několik rozhraní pro přístup k XML datům, například DOM, SAX, JAXP, StaX, .NET XML a další. Nicméně každé rozhraní má své výhody a nevýhody. Tato práce analyzuje existující XML rozhraní a identifikuje jejich pro a proti z různých úhlů pohledu. Na základě této analýzy je navrženo nové DOM rozhraní pro přístup k příliš velkým XML dokumentům a je vytvořena jeho prototypní implementace. Nové rozhraní využívá metodu dělení vstupního XML dokumentu na menší XML dokumenty a dotazování provádí na těchto menších XML dokumentech. Na závěr je nové rozhraní podrobeno řadě experimentů a je srovnáno s existujícím řešením.

Klíčová slova: XML, SAX, DOM, XML rozhraní

Title: Advanced Interface for XML Data

Author: Jiří Korchňák

Department: Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková, PhD.

Supervisor's e-mail address: irena.mlynkova@mff.cuni.cz

Abstract:

Currently there are several interfaces for XML data, such as DOM, SAX, JAXP, StaX, .NET XML support etc. However, each of them has its advantages and disadvantages. This work analyses the existing XML APIs and identifies their pros and cons from various points of view. On the basis of the results a new DOM API for large XML data is proposed and is provided its prototype implementation. New API uses splitting method for input XML document to smaller ones and querying performs on these smaller XML documents. Finally, using a set of experiments the newly proposed API is compared with the existing ones.

Keywords: XML, SAX, DOM, XML interface

1 Introduction

XML (Extensible Markup Language), standardized in [W3C: XML], is a frequently used data format for representing, exchanging and manipulating data. XML can be found as a supplement in HTML (HyperText Markup Language) code [HTML 4.0], settings files of various applications, body of SOAP (Simple Object Access Protocol) messages [WRC: SOAP 1.1] and so on. Because of that, there are many techniques how to process XML data.

In this thesis we first analyse the most known APIs (Application programming interfaces) for process XML and evaluate their advantages and disadvantages.

The main goal of the thesis is to propose a new API for XML data on the basis of previous analysis, which will combine the best of them (main aim is to combine DOM and SAX standards). This proposed API will be then evaluated by a set of experiments and compared with existing ones.

1.1 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 summarizes used technologies and APIs for accessing and modifying an XML document.

Chapter 3 describes other techniques of using technologies described in Chapter 2. These techniques solve specific problems and deficiencies while XML documents are process by APIs from Chapter 2.

Chapter 4 proposes an API for processing very large XML documents. This API is DOM based and develops technique mentioned in Chapter 3.

An implementation of proposed API using the .NET XML framework is described in Chapter 5.

Chapter 6 then displays experiments results used for performance evaluation of API described in Chapter 4 and implemented in Chapter 5.

2 Existing XML APIs

There are many ways how to access XML data. Some of them are only proprietary solutions for specific applications, but others are generally used as a standard for accessing XML documents. In this chapter we describe in detail the most widely used non-proprietary APIs.

2.1 DOM

DOM (Document Object Model) [W3C: DOM] is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of the processing can be incorporated back into the presented document.

In DOM, the XML document is represented as a graph in the application memory which consists of nodes that represent elements, attributes and other XML constructs. An example of an XML document and its DOM tree-structure is depicted in Figure 2.1 and 2.2.

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <a href>My link</a>
    <h1>My header</h1>
  </body>
</html>
```

Figure 2.1: XML Example

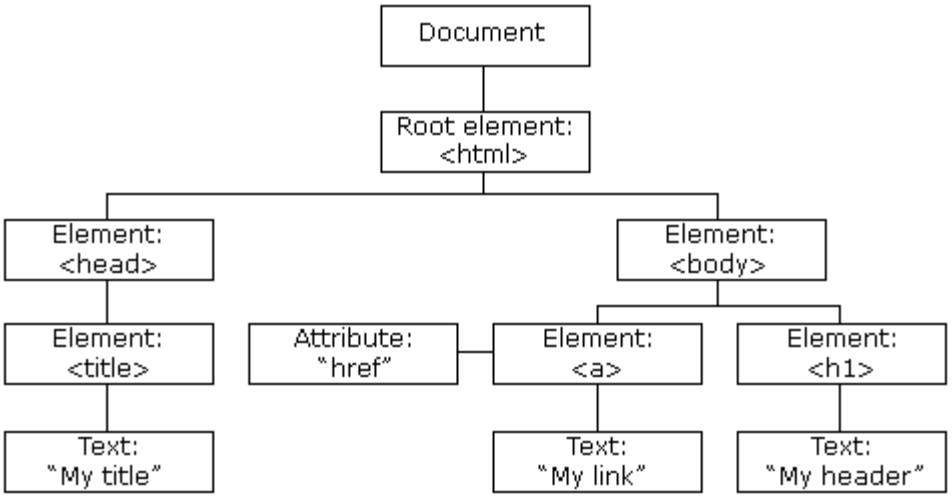


Figure 2.2: DOM Tree of XML Example

There are many levels of DOM. These levels are called DOM Levels and a summary of them can be found in [W3C: DOM Activity]. In the following subchapters we describe every particular level of DOM.

2.1.1 Level 0

DOM Level 0 is not a [W3C] (World Wide Web Consortium) specification. It is just a definition of the functionality equivalent to that found in Netscape Navigator 3.0[1] and Microsoft Internet Explorer 3.0[2]. No independent standard was developed for DOM Level 0, but it was partly described in [HTML 4.0].

This level was created for web browsers that need to access HTML elements such as forms, images, links and anchors[3].

2.1.2 Intermediate DOM

Intermediate DOM is also a definition of the functionality equivalent to that found in version 4.0 of Netscape Navigator and Internet Explorer, adding support for Dynamic

[1] Web browser released by Microsoft in August 13, 1996.

[2] Web browser released by Netscape Communications in August 19, 1996.

[3] These elements were accessible in Netscape Communicator 3.0.

HTML [MS: DHTML], functionality enabling changes to a loaded HTML document. DHTML required extensions to the rudimentary document object that was available in the DOM Level 0 implementations. Although the DOM Level 0 implementations were largely compatible, the DHTML DOM extensions were developed in parallel by each browser maker and remained incompatible.

The Intermediate DOMs enabled manipulation of Cascading Style Sheet [W3C: CSS] properties which influence the visualization of a document. Because of the fundamental incompatibilities in the Intermediate DOMs, cross-browser development required special handling for each supported browser.

2.1.3 Level 1

DOM Level 1 [W3C: DOM Level 1] provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them. The DOM Level 1 Specification is separated into two parts: Core and HTML.

The Core DOM Level 1 section [W3C: DOM Level 1 Core] provides a low-level set of fundamental interfaces that can represent any structured document, as well as defining extended interfaces for representing an XML document. These extended XML interfaces need not be implemented by a DOM implementation that only provides access to HTML documents; all of the fundamental interfaces in the Core section must be implemented. A compliant DOM implementation that implements the extended XML interfaces is required to implement also the fundamental Core interfaces, but not the HTML interfaces.

The HTML Level 1 section [W3C: DOM Level 1 HTML] provides additional, higher-level interfaces that are used with the fundamental interfaces defined in the Core Level 1 section to provide a more convenient view of an HTML document. A compliant implementation of the HTML DOM implements all of the fundamental Core interfaces as well as the HTML interfaces.

2.1.4 Level 2

DOM Level 2 specification is separated into several parts: Core, Views, Events, Style, Traversal and Range and HTML. Each part is specified in its own W3C Recommendation.

The DOM Level 2 Core [W3C: DOM Level 2 Core] is made of a set of core interfaces to create and manipulate the structure and contents of a document. The Core also contains specialized interfaces dedicated to XML and builds on the DOM Level 1 Core.

The DOM Level 2 Views [W3C: DOM Level 2 Views] defines an interface that allows programs and scripts to dynamically access and update the content of a representation of a document and is based on the DOM Level 2 Core.

The DOM Level 2 Events [W3C: DOM Level 2 Events] defines an interface that gives to programs and scripts a generic event system. The DOM Level 2 Events exploits the DOM Level 2 Core and Views.

The DOM Level 2 Style [W3C: DOM Level 2 Style] defines an interface that allows programs and scripts to dynamically access and update the content and of style sheets documents. The DOM Level 2 Style is based on the DOM Level 2 Core and Views.

The DOM Level 2 Traversal and Range [W3C: DOM Level 2 Traversal and Range] contains specialized interfaces dedicated to traversing the document structure and identifying and manipulating a range in a document. The DOM Level 2 Traversal and Range are based on the DOM Level 2 Core.

The DOM Level 2 HTML [W3C: DOM Level 2 HTML] defines an interface that allows programs and scripts to dynamically access and update the content and structure of [HTML 4.01] and [XHTML 1.0] documents. The DOM Level 2 HTML exploits the DOM Level 2 Core and is not backward compatible with DOM Level 1 HTML.

2.1.5 Level 3

DOM Level 3 specification is, also like DOM Level 2, separated into several parts: Core, Load and Save, Validation, XPath, Views and Formatting, Events and Abstract Schemas. Each part is specified in its own W3C Recommendation.

The DOM Level 3 Core [W3C: DOM Level 3 Core] enhances DOM Level 2 Core by completing the mapping between DOM and the XML Infoset [W3C: XML Information Set], including the support for XML Base [W3C: XML Base], adding the ability to attach user information to DOM Nodes or to bootstrap a DOM implementation, providing mechanisms to resolve namespace prefixes or to manipulate "ID" attributes, giving to type information, etc. The DOM Level 3 Core is based on the DOM Level 2 Core.

The DOM Level 3 Load and Save [W3C: DOM Level 3 Load and Save] defines an interface that allows programs and scripts to dynamically load the content of an XML document into a DOM document and serialize a DOM document into an XML document; DOM documents being defined in [W3C: DOM Level 2 Core] or newer, and XML documents being defined in [W3C: XML] or newer. It also allows filtering of content at load time and at serialization time.

The DOM Level 3 Validation [W3C: DOM Level 3 Validation] provides a guidance to programs and scripts to dynamically update the content and the structure of documents while ensuring that the document remains valid, or to ensure that the document becomes valid.

The DOM Level 3 XPath[4] [W3C: DOM Level 3 XPath] provides simple functionalities to access a DOM tree using [W3C: XPath 1.0].

The DOM Level 3 Views and Formatting [W3C: DOM Level 3 Views and Formatting] defines an interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The DOM Level 3 Views and Formatting is based on the DOM Level 2 Views.

The DOM Level 3 Events [W3C: DOM Level 3 Events] defines a generic event system which allows registration of event handlers, describes event flow through a tree structure, and provides basic contextual information for each event. The DOM Level 3 Events is based on the DOM Level 2 Events.

The DOM Level 3 Abstract Schemas [W3C: DOM Level 3 Abstract Schemas] defines an interface that allows programs and scripts to dynamically access and update the content, structure and style of documents.

[4] XPath is a language for addressing parts of an XML document.

2.2 SAX

SAX (Simple API for XML) [SAX] is widely-used specification[5] that describes how XML parsers can pass information efficiently from XML documents to software applications. According to the specification, SAX is a sequential access parser API for XML. By another name SAX is an event-based API that reports parsing events (such as the start and end of elements) directly to the application through callbacks, and does not usually build an internal tree.

The SAX events include:

- XML Text nodes
- XML Element nodes
- XML Processing Instructions
- XML Comments

It is important to know that SAX parsing is unidirectional; previously parsed data cannot be re-read without starting the parsing operation again.

To understand how SAX can work, consider sample document in Figure 2.3.

```
<?xml version="1.0"?>
<book>
  <paragraph>Hello, world!</paragraph>
</book>
```

Figure 2.3: Sample XML document

An event-based interface will break the structure of document in Figure 2.3 down into a series of linear events that are given in Figure 2.4.

[5] There is no formal specification for SAX, but its [Java] implementation is considered to be normative.

```
start document
start element: book
start element: paragraph
characters: Hello, world!
end element: paragraph
end element: book
end document
```

Figure 2.4: Fired SAX events on sample document

If the sample document had attributes, then with start element event fired on such element an attribute list structure is given. User can iterate through an attribute list using simple AttributeList interface.

2.3 StAX

StAX (Streaming API for XML) [StAX] is a bi-directional API for reading and writing XML data. Until StAX there were only two main approaches to process an XML document: 1) SAX and 2) DOM. StAX was designed as a median between these two solutions.

StAX consists of two styles: A low-level cursor API (see Chapter 2.3.1), designed for creating object models and a higher-level event iterator API (see Chapter 2.3.2), designed to be used in pipelines and be easily extensible.

Both APIs can be thought of as iterating over a set of events. In the cursor API the events may not be performed; in the event iterator API the events are performed. Both APIs break down an XML document into the following events:

- StartElement
- EndElement
- Attribute
- Namespace
- Characters
- EntityReference
- ProcessingInstruction
- Comment
- StartDocument
- EndDocument
- DTD
- NotationDeclaration
- EntityDeclaration

2.3.1 Cursor API

The cursor API [StAX: Cursor API] moves a virtual cursor across the XML input and provides accessor methods to the contents pointed to by the virtual cursor. The cursor API is designed to be an efficient and low-level mean for constructing object models and representations of the XML documents. The cursor can point to one thing at a time, and always moves forward. The cursor model is supported by the XMLStreamReader and XMLStreamWriter interfaces.

An example of using XMLStreamReader and XMLStreamWriter interfaces in Java[6] language is given in Figure 2.5 or more precisely 2.6. The result of operation in Figure 2.6 is given in Figure 2.7.

```
XMLInputFactory f = XMLInputFactory.newInstance();
XMLStreamReader r = f.createXMLStreamReader( ... );
while(r.hasNext()) {
  r.next();
}
```

Figure 2.5: Instantiation of an input factory, creating a reader and iterating over the elements of an XML document

```
XMLOutputFactory output = XMLOutputFactory.newInstance();
XMLStreamWriter writer = output.createXMLStreamWriter( ... );
writer.writeStartDocument();
writer.setPrefix("c","http://c");
writer.setDefaultNamespace("http://c");
writer.writeStartElement("http://c","a");
writer.writeAttribute("b","blah");
writer.writeNamespace("c","http://c");
writer.writeDefaultNamespace("http://c");
writer.setPrefix("d","http://c");
writer.writeEmptyElement("http://c","d");
writer.writeAttribute("http://c","chris","fry");
writer.writeNamespace("d","http://c");
writer.writeCharacters("foo bar foo");
writer.writeEndElement();
```

Figure 2.6: Instantiation of an output factory, creating a writer and writing XML output.

[6] Java is a programming language originally developed at Sun Microsystems and released in 1995.

```
<?xml version='1.0' encoding='utf-8'?>
<a b="blah" xmlns:c="http://c" xmlns="http://c">
<d:d d:chris="fry" xmlns:d="http://c"/>foo bar foo</a>
```

Figure 2.7: Result of operations in Figure 2.6

2.3.2 Event Iterator API

The Event Iterator API [StAX: Event Iterator API] is easy-to-use and extend. It involves a set of events that an application programmer can use to manipulate XML data. The events are allocated and no restrictions are placed on their reuse. The events are designed to be easy-to-filter, buffer, persist and compare. For iterating over XML contents and manipulating of XML files are used XMLEventReader and XMLEventWriter interfaces.

An example of reading all the events on a stream and printing them is given in Figure 2.7.

```
while(stream.hasNext()) {
  XMLEvent event = stream.nextEvent();
  System.out.print(event);
}
```

Figure 2.7: Event Iterator API in action

2.4 JAXP

JAXP (Java API for XML Processing) [JAXP] is a complex Java library for XML processing. JAXP enables applications to parse, transform, validate and query XML documents using an API that is independent of a particular XML processor implementation. JAXP supports a number of different standards to process XML documents:

- DOM
- SAX

- StAX[7]
- XSLT [W3C: XSLT]

An example of JAXP architecture is given in Figure 2.8.

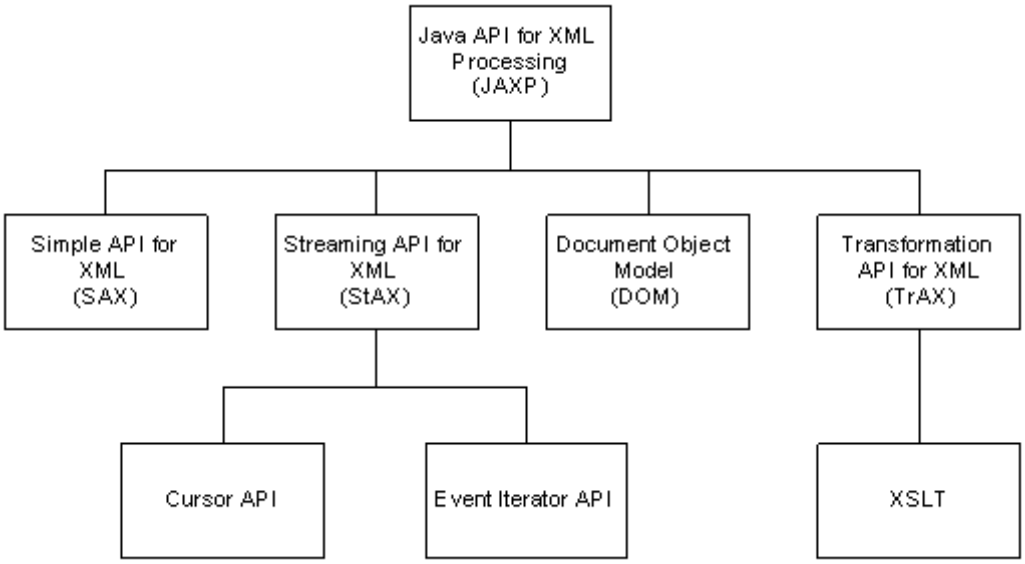


Figure 2.8: JAXP architecture

JAXP allows for using any XML-compliant parser from within user's application. The layer that allows for it is called pluggability layer and lets a user to plug in an implementation of the SAX, StAX or DOM API. The pluggability layer also allows a user to plug in an XSL processor, letting him/her to control how his/her XML data is displayed.

2.5 .NET XML

.NET XML [XML in .NET] is – like JAXP for Java – a complex .NET library for XML processing. The .NET XML framework provides a support for all of the W3C XML specifications including:

- XML 1.0
- Namespaces in XML [W3C: XML Namespaces]
- DOM Level 2

[7] StAX is supported from version 1.4 of JAXP.

- XPath 1.0 [W3C: XPath 1.0]
- XSLT 1.0
- XML Schema [W3C: XML Schema][8]

In addition to the W3C specifications, the .NET XML framework provides a support for several other XML-related technologies that do not share the same level of acceptance but which are quickly gaining mind-share as they continue to evolve.

Streaming XML APIs (like SAX):

- SOAP 1.1
- WSDL [W3C: WSDL 1.1]
- UDDI [OASIS: UDDI]

At the core of the .NET Framework XML classes there are two abstract classes: *XmlReader* and *XmlWriter*. XmlReader provides a fast, forward-only, read-only cursor for processing an XML document stream. XmlWriter provides an interface for producing XML document streams that conform to [W3C: XML]. There are three concrete implementations of XmlReader - XmlTextReader, XmlNodeReader, and XslReader - as well as two concrete implementations of XmlWriter: XmlTextWriter and XmlNodeWriter. XmlTextReader and XmlTextWriter support reading from and writing to a text-based stream, while XmlNodeReader and XmlNodeWriter are designed for working with in-memory DOM trees (see Figure 2.9).

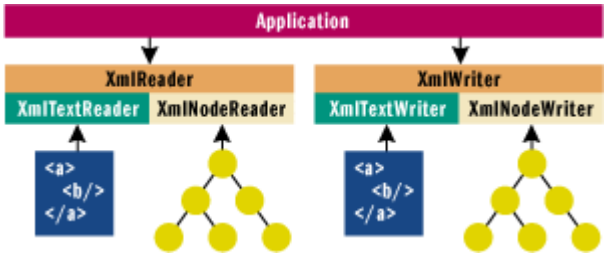


Figure 2.9: XmlReader and XmlWriter class

[8] Refers to [W3C: XML Schema - Datatypes] and [W3C: XML Schema - Structures]

2.6 Resume

Some of the most widely used APIs are described in previous subchapters and now we try to evaluate them.

DOM

Time

+ Can access widely separately parts of the document at the same time.
+ Building a complete tree may be time consuming.

Memory

+ Stores the entire XML document into memory.
+ Problem when input XML is large than available memory.

User-friendliness

+ Contains a lot of functions due to the tree representation of XML document.
+ Easy insertion/deletion of new/old tree node.
+ Preserves comments.
+ Can traverse in any direction in XML.

Typical uses of DOM API is in application where we need to modify the document repeatedly, to use an internal data structure which is almost as complicated as the document itself or to store the document for a significant amount of time through many method calls.

SAX

Time

+ Parser is relatively fast.
− Can access only one event at the time.

Memory

+ Does not store XML into memory.
+ Can access a very large XML documents.

User-friendliness

+ Only simple API, easier to teach.
± Traverses from top to bottom.

– Cannot preserve comments.
– Cannot insert/delete node.
– Only simple API, users have to take care of more (data structures etc.).

Typical uses of SAX API is in application where we need to read data quickly (for example in SOAP headers) or where we do not need to our own data structures as complicated as the DOM tree.

StAX

Time

+ Parser is relatively fast.

Memory

+ Does not store XML into memory.
+ Can access a very large XML documents.

User-friendliness

+ Pull parser.
+ Two distinct APIs (Cursor, Iterator).
+ Bidirectional API enabling both reading and writing of XML documents.
+ Clients can read multiple XML documents simultaneously.
+ Parser can be stopped.
+ Preserves comments.
± Traverses from top to bottom.
– Only simple APIs, users have to take care of more (data structures etc.).

Typical use of StAX API is in application where we need to change data quickly (for example in headers of SOAP messages on SOAP intermediary).

JAXP, .NET XML

+ Contains all basic APIs (DOM, SAX, StAX).

Typical use of JAXP is in application written in Java language and typical use of .NET XML is in application written in all of .net supporting languages. Developers using these libraries can choose what particular API they wanted to use.

3 Other Techniques

APIs described in the previous chapter have many uses, but in some cases the usually used techniques are insufficient (for example building a DOM tree of a very large XML file may end in run-out of memory…). The other techniques come in with solutions for these cases and in this chapter we describe in detail some of them.

3.1 A DOM Method to Retrieve Data from a Very Large XML Document

[KYH DOM] describes a DOM method for retrieving data from a very large XML document with manageable memory space and processing time by a single general-purpose personal computer.

A very large XML document is partitioned into *n* small documents, where *n* varies depending on the capacity of the given resource such as a personal computer. Each of the *n* small documents is then modified by a padding process to meet the well-formedness of the XML document. A data retrieval operation on the original large XML document, which is expressed with DOM API, is then executed sequentially on the small XML tree that is built from each of the modified *n* XML documents, and the results from all the *n* XML trees are combined to generate the final result. With this approach, the data retrieval operations on the very large XML document can be executed by a single general-purpose personal computer.

Partitioning, padding and retrieving are each explained in further detail below.

3.1.1 Partitioning

Definition: *Let D be a document given by a sequence of characters. If each of n documents, F_1, F_2, …, and F_n, is a subsequence of D such that the concatenation of F_1, F_2,…, and F_n is equal to D, then a sequence of the n documents, $<F_1, F_2, …, F_n>$, is a* ***partition*** *of a document D, where each F_i, i=1,…,n, is called a* ***fragment*** *of D.*

In explaining how to partition the given XML document, the authors use some notations defined as follows:

1. *The size of a document D is expressed as D.length.*
2. *The i-th character of a document D is expressed as D[i] ($1 \leq i \leq D.length$).*
3. *The sequence of characters from i-th character to j-th character of a document D is expressed as D[i, j]. Therefore, the contents of a document D are expressed as D[1, D.length].*
4. *The document generated by concatenating two documents D_1 and D_2 is expressed as $D_1 \cdot D_2$.*

Definition: *Boundary between fragments of an XML document is called the* ***cut point****.*

How partition process works is formally described in Listing 1.

Algorithm 1: ***PARTITION(D, n, h)***

<Input>
1. *D* : An XML document to be partitioned
2. *n* : The expected number of fragments to be generated
3. *h* : The width of a range for finding a cut point

<Output>
1. *n'* : The actual number of fragments generated
2. $<F_1 F_2 \ldots F_{n'}>$: A partition of D ($1 \leq n' \leq n$)
3. $<C_1 C_2 \ldots C_{n'-1}>$: A sequence of cut points
4. *P* : The prolog of *D*

```
1  Declare StackOfStartTags as a stack of strings;
2  Declare Ts, Tnew as variables pointing to start- or end-tags;
3  TargetSize := ⌈D.length / n⌉;
4  Initialize StackOfStartTags;
5  Prolog := the prolog of D;
6  StartPoint := the offset of the end of prolog of D + 1;
7  CutPoint := StartPoint;
8  i := 1;
9  While ((StartPoint < D.length) and (i < n)) do
10     InitCutPoint := StartPoint + TargetSize;
11     From := CutPoint;
12     While (there is a tag in [From, D.length] of D) do
13         Scan a tag and set Ts to point to the tag retrieved;
14         If (Ts points to a start-tag)
15             Push the name of the pointed start-tag by Ts into StackOfStartTags;
16             If (the start-tag pointed by Ts is in [InitCutPoint – h/2, D.length] of D)
17                 jump FindCutPoint;
18         End if
19         If (Ts points to an end-tag)
```

```
20          Pop the last item from StackOfStartTags;
21      From := (the offset of the end of the pointed tag by Ts)+1;
22  End while
23  FindCutPoint:
24  minimalDistance := the number of items in StackOfStartTags;
25  CutPoint:= (the offset of the end of the pointed tag by Ts) + 1;
26  Scan a tag and set Tnew to point to the tag retrieved;
27  While (the tag pointed by Tnew is in [CutPoint, InitCutPoint + h/2] of D) do
28      If (Tnew points to a start-tag)
29          Push the name of the pointed tag by Tnew into StackOfStartTags;
30          Distance := the number of items in StackOfStartTags;
31          If (minimalDistance > Distance)
32              minimalDistance:= Distance;
33              Ts:= Tnew;
34              CutPoint:= (the offset of the end of the pointed tag by Ts)+1;
35          End if
36      End if
37      If (Tnew points to an end-tag)
38          Pop the last item from StackOfStartTags;
39      Scan a tag and set Tnew to point to the tag retrieved;
40  End while
41  Fi := D[StartPoint, CutPoint];
42  StartPoint := CutPoint + 1;
43  Ci := CutPoint;
44  i := i + 1;
45 End while
46 Fi := D[StartPoint, Do.length];
47 n':= i;
48 Return n', <F1 F2 ... Fn' >, <C1 C2 ... Cn' > and Prolog;
```

3.1.2 Padding

Since each fragment generated in the partitioning step does not comprise a well-formed XML document, it is modified into a well-formed XML document in the padding step.

Definition: *Let e_{i1} and e_{in} be two nodes of an XML tree where e_{i1} is an ancestor of e_{in}. If the sequence of nodes in the path from e_{i1} to e_{in} is given by $e_{i1}e_{i2}\ldots e_{i(n-1)}e_{in}$, then*

*(1) The **front pad** from e_{i1} to e_{in} is a string given by*

$$<e_{i1}><e_{i2}>\ldots<e_{i(n-1)}>$$

*(2) The **back pad** from e_{i1} to e_{in} is a string given by*

$$</e_{i(n-1)}>\ldots</e_{i2}></e_{i1}>$$

Some other notations that authors used for explaining the padding algorithm are defined as follows:

1. *The **cut element**, split at the position of C_i into two fragments, is expressed as CutElement(C_i).*
2. *The first start-tag of a fragment F is expressed as F.first.*
3. *The last end-tag of a fragment F is expressed as F.last.*
4. *The root of an XML document D is expressed as D.root.*
5. *The front pad from e_{i1} to e_{in} is expressed as FPad(e_{i1}, e_{in}).*
6. *The back pad from e_{i1} to e_{in} is expressed as BPad(e_{i1}, e_{in}).*

Definition: *A **cut attribute** is an attribute that is added to each start-tag <e_i> of a cut element.*

Definition: *For differentiating the elements, which are generated by padding algorithm, from original and cut elements, an attribute known as a **dummy attribute** is added to each start-tag of these generated elements. An element that has a dummy attribute in a start-tag is a **dummy element**.*

Algorithm 2: ***PAD(D, <F_1 F_2 …F_n>, <C_1 C_2 …C_n>, P)***

<Input>
1. *D* : An XML document
2. <F_1 F_2 ...F_n> : A partition of *D*
3. <C_1 C_2 ...C_n> : A sequence of cut points
4. *P* : The prolog of *D*

<Output>
1. <D_1 D_2 ...D_n> : A sequence of XML documents

```
1  Declare <F'1 F'2 ... F'n>, <F''1 F''2 ... F''n> as sequences of documents;
2  For each Fi from i=1 to n
3      If (Fi is not the first fragment of <F1 F2 ...Fn>)
4          StartTag :=Duplicate the start-tag of CutElement(Ci-1);
5          Add a cut attribute to StartTag;
6          Add an identification attribute to StartTag;
7          F'i:= StartTag ·F'i;
8      Else
9          F'i:= Fi;
10     End if
11     If (Fi is not the last fragment of <F1 F2 ...Fn>)
12         EndTag := Generate the end-tag of CutElement(Ci);
13         F'i := F'i·EndTag;
14         Add a cut attribute to the corresponding start-tag of F'i.last;
15         Add an identification attribute to the corresponding start-tag of F'i.last;
16     End if
17     If (Fi is not the first fragment of <F1 F2 ...Fn>)
18         Front := Compute FPad(D.root, CutElement(Ci-1));
19         For each start-tag in Front
```

```
20            Add an identification attribute to the start-tag;
21            Add a dummy attribute to the start-tag;
22        End for
23    End if
24    If (Fi is not the last fragment of <F1 F2 ...Fn>)
25        Rear := Compute BPad(D.root, CutElement(Ci));
26    F''i := Front · F'i · Rear;
27 End for
28 For each F''i from i=1 to n
29    Di:= P · F''i;
30 End for
31 Return <D1 D2 ... Dn>;
```

3.1.3 Retrieving

The data retrieval operation is expressed with DOM API, and this operation is based on an assumption that the XML tree of an original large XML document exists in the main memory.

Authors selected the following GET operations from DOM API as the representatives of data retrieval operations.

Interface	Operation Name	Description
Document	*getElementsByTagName*	Returns a nodelist of all the elements in document order with a given tag name.
Node	*getChildNodes*	Returns a nodelist that contains all children of this node.
	getFirstChild	Returns the first child of this node.
	getLastChild	Returns the last child of this node.
Element	*getElementsByTagName*	Returns a nodelist of all descendant elements with a given tag name, in document order.

Table 3.1: The DOM API data retrieval operations

The operations in Table 3.1 have a difference in external forms. However, each of these *GET* operations can be considered to receive some nodes that satisfy the specified condition from the given XML tree, *T*. Thus, it suffices to show how to implement the operation of *GET(T,e,P,S)* which returns the nodes that satisfy the condition *P* from a subtree of *T*, having *e* as its root. The fourth parameter *S* specifies how to make the result of the operation from the retrieved data. The value of *S* can be one of three values, *ALL*, *FIRST* and *LAST*. Let N_r be the nodes satisfying the condition *P* from a subtree of *T*,

having *e* as its root. If the value of *S* is *ALL*, *GET* operation returns N_r as a list of nodes. Yet, if the value of *S* is *FIRST* or *LAST*, *GET* operation returns the first or last node of N_r correspondingly.

For example, the operation of *getElementsByTagName(XXX)* with *Document* interface on an XML tree *T* can be given by *GET (T, the root of T, tagname=XXX, ALL)* and the operation of *getFirstChild()* with *Node e*, interface on an XML tree *T* can be given by *GET(T, e, child of e, FIRST)*.

The retrieval procedure of *GET(T,e,P,S)* can be summarized by the following algorithm *RETRIEVE(D,e,P,S)*, where *T* is the XML tree of the given XML document *D*. The following algorithm *UNIFY(e)* is called to unify the split child nodes of node *e*.

Algorithm 3: ***RETRIEVE(D, e, P, S)***

<Input>
1. *D* : An XML document from which data is to be retrieved
2. *e* : An element of *D*
3. *P*: A condition that can be satisfied by an element of *D*
4. *S*: one of three values, *ALL*, *FIRST* and *LAST*

<Output>
1. N_r: A node list that is the same result of the *GET(T, e, P, S)*.

Declare T_r as an XML Tree having only a root node;
Declare N_r as an empty node list;
Declare DP as a general- purpose DOM parser;
Generate a sequence of multi-small XML documents <$D_1D_2...D_n$> by using algorithms PARTITION and PAD on D;
For each *D_i from i=1 to n*
    T_i := Build an XML tree from D_i by using DP;
    N_i := Execute GET(T_i, e, P, S) by using DP;
    For each *node e of N_i*
        If *((e is a cut element) or (e is a dummy element))*
            e_t := Find the original element of e from N_r by using the identification attribute of e;
            If *(e_t exists)*
                Copy the contents of e into e_t by using DP;
            Else
                Copy e as a child of T_r by using DP;
                Add e into N_r using P;
            End if
        Else
            Copy e as a child of T_r by using DP;
            Add e into N_r by using DP;
        End if
    End for
End for
Remove cut attributes from every node e of N_r;
Return *N_r;*

Algorithm 4: ***UNIFY(e)***

<Input>
1. *e*: An element
<Output>
1. *Unified e*: An element having no dummy and cut child elements

```
1  For each child element e_c of e
2      If ((e_c is a cut element) or (e_c is a dummy element))
3          e_t := Find the original element of e_c from all child elements of e by using the identification attribute of e_c;
4          If (e_t exists)
5              Copy the contents of e_c into e_t;
6              Remove the link from e to e_c;
7          End if
8      End if
9  End for
10 Return e;
```

The authors apply a lazy approach to unify the split of child nodes: the first time a node *e* of a node list is referred by an XML application, the child nodes of *e* having a dummy or cut attribute and the same identifier are combined. This can avoid time consumption caused by complete navigation of subtrees for unifying unused nodes of T_r.

3.2 Hybrid Parallelism for XML SAX Parsing

[PZC SAX] is a technique that combines pipelined and data parallelism to form a hybrid SAX parser to achieve a faster parsing time of an input XML document. This technique uses four-stage software pipeline with a combination of sequential and data-parallel stages[9].

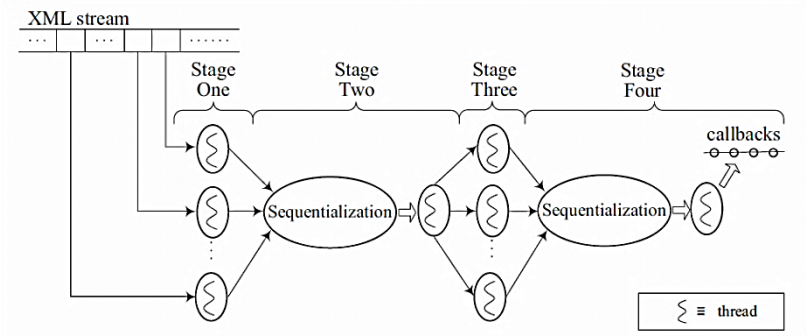


Figure 3.1: The pipeline design of a Hybrid Parallel SAX Parser

[9] Particular stages are described in following subchapters.

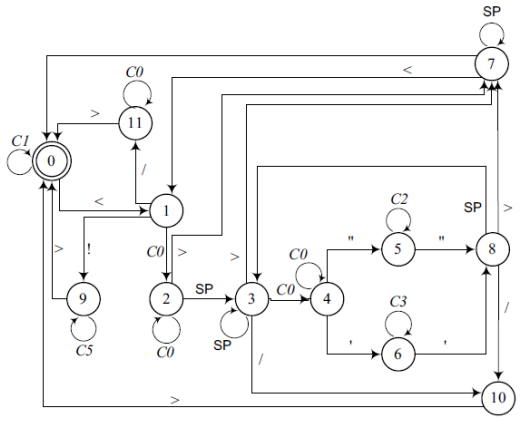
From Figure 3.1 it is obvious that Stage One and Stage Three are data parallel and Stage Two and Stage Four are sequential.

3.2.1 Stage One - Preliminary Parsing

The purpose of Stage One is to identify the basic structure of the XML stream. It determines the start-tags, element contents, and end-tags. Within the start-tags, it determines the start and end character positions of the element name and each attribute (including namespace declarations), and determines the start and end positions of them. Within end-tags, the start and end character positions of the element name is determined.

When the XML stream is received, it is divided up into chunks. Parallelism is obtained by parsing multiple chunks independently and in parallel, so this stage is an example of data parallelism. But a preliminary parser that begins parsing at some arbitrary point in an XML stream (without having seen all previous characters) will not know on which state to begin. To address this, authors use [Meta-DFA].

Stage One preliminary parser is a DFA[10] with actions on transitions. Next a *product* machine is built from the original DFA which executes multiple copies of the original DFA simultaneously. For each state *q* of the original DFA, the meta-DFA includes a complete copy of the DFA as a sub-DFA which begins execution in state *q* at the beginning of the chunk. Each copy thus begins parsing the chunk in a different state of the original DFA. The meta-DFA thus essentially pursues simultaneously all possible guesses as to the state at the beginning of the chunk. Since the meta-DFA is also a DFA, the simultaneity is strictly conceptual, and can still be executed by a single core.



C0 ≡ any unicode character except <, >, /, !, ", '
C1 ≡ >, /, !, ", ' and C0 defined above
C2 ≡ <, /, !, ' and C0 defined above
C3 ≡ <, /, !, " and C0 defined above
C4 ≡ >, /, !, ", ' and C0 defined above
C5 ≡ <, /, !, ", ' and C0 defined above

Figure 3.2: The DFA for Stage One

[10] Deterministic Finite-State Automaton.

3.2.2 Stage Two - Chunk Resolution

Multiple sequences of structural parts such as start-tags are attached to each chunk at this point. The purpose of Stage Two is to resolve these ambiguous results into a single, unambiguous sequence and also resolves split units[11], completing them so that they can be processed normally.

3.2.3 Stage Three - Namespace Processing

With position information from Stage Two, Stage Three performs additional namespace processing. After such processing, the required data for the XML SAX callbacks are largely present, so Stage Three works as a stage which will turn syntactic units and their position information on the XML stream into buffered data with the required representation forms for the XML SAX callbacks. The output of Stage Three consists of structures that will eventually be used in the callbacks.

Stage Three resolves the element names into a namespace prefix and a local name. DFA used by authors is shown in Figure 3.3. Any character sequence ending in state 1 is confirmed as an *UnprefixedName* sequence. Any character sequence ending in state 2 is a *PrefixedName*, with the *Prefix* given by the sequence before the colon, and the *LocalPart* given by the character sequence after the colon.

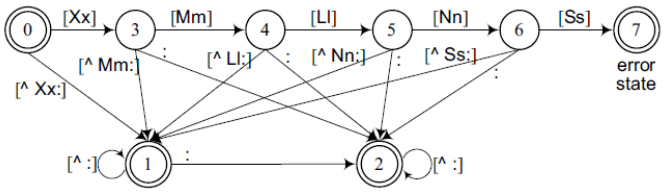


Figure 3.3: The DFA for parsing element names[12].

Stage Three also identifies namespace declarations and attributes, and, at the same time, it will resolve namespace declarations into namespace prefix and URI pairs and each

[11] Some XML syntactic units split across two chunks. This means that one chunk will contain the unit's start position, and the other chunk will contain its end position.

[12] A transition is taken when the accepting character satisfies the regular expression shown on the transition edge.

attribute into namespace prefix, local name, and value. The DFA used by authors is shown in Figure 3.4. State 0 is the start state and state 19 is the end state.

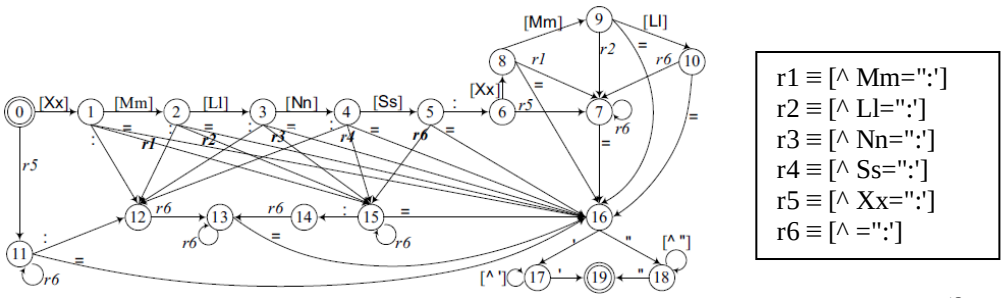


Figure 3.4: The DFA for recognizing and parsing attributes and namespace declarations[13].

Stage Three will also resolve prefixes to namespace URIs if their corresponding namespace declarations are within the current chunk. Since Stage Three works on chunks independently in parallel, the prefix lookup scope of any given chunk is limited to that chunk. Some namespace prefixes will thus refer to namespace declarations in other chunks. Therefore, some prefixes cannot be resolved to URIs in Stage Three. These unresolved prefixes will be resolved in Stage Four.

3.2.4 Stage Four – Callbacks

Stage Four resolves any remaining inter-chunk namespace prefix references, and invokes the actual callbacks for the SAX events. This stage is sequential, different threads may be in stage four, but never more than one thread at any time.

3.2.5 Execution

How authors parsing algorithm work in four steps is described in this section. At the beginning of execution, a thread is started for every core. At any time, each thread will be working on a chunk in some stage, but a chunk may be processed by different threads at different stages. When a chunk is ready for the next stage, it will be processed by the next available thread.

[13] A transition is taken when the accepting character satisfies the regular expression shown on the transition edge.

Algorithm 5: *PARSING*

<Input>

1. *T[]*: Threads started for every core and working on a chunk in some stage

STEP 1:

```
If processing is finished
    Exit;
Else
    If T[] contains a thread in Stage Four
        Go to STEP 2;
    Else
        If there are any chunks that need Stage Four processing
            Perform Stage Four processing on all chunks that are ready for Stage Four;
                {During Stage Four processing, once a chunk's callbacks are finished, the current thread is read in a new chunk
                into the buffer space that was made available by the completion of the chunk.}
            Go to STEP 2;
        Else
            Go to STEP 2;
        End if
    End if
End if
```

STEP 2:

```
If T[] contains a thread in Stage Two
    Go to STEP 3;
Else
    If there are any chunks which need Stage Two processing
        Start from the first chunk that is ready for Stage Two, and proceed as far as possible, until
        encountering a chunk still in Stage One;
        Go to STEP 3;
    Else
        Go to STEP 3;
    End if
End if
```

STEP 3:

```
If there are chunks which need Stage Three processing
    Take the first such chunk and perform Stage Three on it;
    Go to STEP 1;
Else
    Go to STEP 4;
End if
```

STEP 4:

```
If there are chunks which have been read in, but are not yet undergoing Stage One processing
    Take the first such chunk and start Stage One processing on this chunk;
    Go to STEP 1;
Else
    Go to STEP 1;
End if
```

3.3 Lazy XML Parsing/Serialization based on Literal and DOM Hybrid Representation

[TaT DOM] proposes a method for efficient XML processing of SOAP nodes in distributed SOA computing environments. In particular, on a SOAP intermediary[14], the parsing and serialization are inverse data model conversions. The inverse conversion is often redundant, but this overhead can be a bottleneck for the performance of the SOAP intermediary. The method removes this inefficient processing by reusing the original literal XML representation as much as possible when the serialization of an XML data is needed. Some layers used in XML processing are shown in Figure 3.5

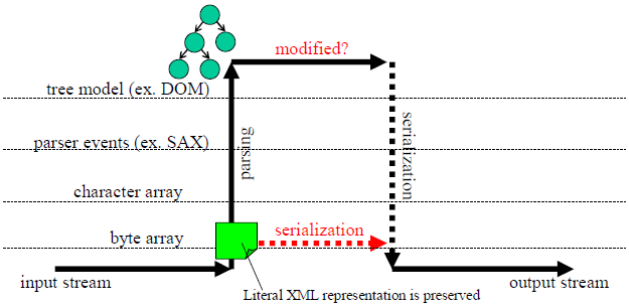


Figure 3.5: Layers in XML processing

3.3.1 Partial XML Parsing and Partial XML Tree Construction

In Literal DOM implementation, only the accessed DOM objects are created. First, the parser creates a *Document* object. The *Document* object internally holds the incoming literal XML document as a byte array. Also, the *Document* object has a *table* for the parsed node information. At this step, the actual DOM object has not been created yet. The actual data, such as element name or text value, is just included in the literal XML byte array. The table holds some *metainformation*, such as the *regions* of the nodes.

In the table, a row represents a node in the XML tree. The table has 4 columns, *type*, *offset*, *length*, and *node*. The *index* represents the unique index of the node, but the *index* column does not actually exist, since that is the row number in the table. The *type* column entries are Document, Element, or Text. The *offset* is the starting position of the node in the literal XML byte array. The *length* is the length of the node, so the length

[14] Intermediaries are entities positioned between a client and service provider that provide additional functionality.

column value is not recorded until the end tag is parsed if the node is an Element. The *node* column has a reference to the DOM node object if any. If the DOM object has not been created yet, the *node* has null value. The order of the nodes in the table is the XML document order. Therefore, the *offset* column always increases.

An working example of described representation can be shown on *Node.getFirstChild()* method called on node *Node* (described in Listing 6).

Algorithm 6: ***Node.getFirstChild()***

<Input>
1. *Node* : Node from that is method *getFirstChild* called

<Output>
1. *OutNode* : First child of *Node* or *null.*

```
1  Declare Index as the index of the Node;
2  Declare offset[], length[], node[]: each column for offset, length, node;
3  Declare OutNode as null;
4  If offset[Index+1] is not filled;
5      read input stream for offset[Index+1];
6  If length[Index] is not filled or offset[Index+1] < offset[Index] + length[Index];
7      If node[Index+1] is null;
8          create new DOM object for node[Index+1];
9      EndIf
10     OutNode := node[Index+1];
11 EndIf
12 Return OutNode;
```

3.3.2 Serialization Using the Original Literal XML Representation

The authors used an existing StAX parser in their Literal DOM implementation. Since StAX parsing can be stopped in the middle of the XML data, partial parsing is possible. By hooking the incoming XML input stream to the StAX parser, the literal XML message can be preserved. Also, the position of each event in literal XML message can be obtained by using the *Location* API in the StAX API.

An existing DOM node implementation was used as the basis of Literal DOM implementation. The Literal DOM wraps the original DOM nodes. A *Document* object in Literal DOM is actually a *DocumentProxy* object (wrapping an original Document object). The *DocumentProxy* object internally holds the incoming Literal XML as a byte array and

a table for the node position information. The *DocumentProxy* also creates each of the DOM node objects.

In serialization, if there is no change history, the literal XML byte array is just used as the serialized form. Otherwise, the literal XML byte array is updated by using change history.

3.3.3 Deferred Update Using the Change History

An entry in the change history is classified into three types: *Insertion*, *Deletion*, or *Substitution*.

When the insertion method, such as Node.appendChild(), is called, an entry is created in the change history, but the literal XML representation is not updated yet. The *Insertion* entry has the inserted new node and its offset. The offset is the position of the insertion in the original literal XML byte array.

When the deletion method, such as Node.removeChild(), is called, an entry is created in the change history, but the literal XML representation is not updated. The *Deletion* entry has an offset and length in the original literal XML byte array. The offset and length is same as the offset and length of the deleted node.

Similar plan takes place, when the substitution method, such as Node.replaceChild(), is called. The *Substitution* entry has the offset and length of the deleted node, and the substituted node object itself. Substitution can be viewed as a combination of deletion and insertion to the same position.

If a new change does not overlap with any previous change, a new change entry is simply added. If a new change is added to an existing entry in the change history, the new change is directly reflected into the node in the existing entry. If a new change region covers some existing entries, the affected existing entries are removed from the change history.

Now, with the knowledge of using change history, the serialization process can be defined (see Listing 7).

Algorithm 7: *Serialization*

<Input>

1. *Entry[]* : Each entry in the change history
2. *Entry[].newChild* : Inserted or substituted node if any

3. *Entry[].offset* : Position of the entry on the literal XML
4. *Entry[].length* : Deleted or substituted length if any

<Output>

1. Serialized XML.

```
1  Declare Current as a current position and set it to 0;
2  For each Entry[i]
3      write literal XML from current to entry[i].offset into output stream;
4      If (Entry[i].newChild exists)
5          serialise Entry[i].newChild into output stream;
6      If (Entry[i].length exists)
7          set Current to Current + Entry[i].length;
8  EndFor
9  Write from Current to the end of the literal XML
```

3.4 Prefiltering Techniques for Efficient XML Document Processing

[XML Prefiltering] proposes a prefiltering framework, where repeated access to a large XML document can be efficiently carried out within the existing DOM and SAX models. The prefiltering framework essentially uses a tiny search engine to locate useful fragments in target XML documents by approximately executing the user's queries. Those fragments are gathered into a candidateset XML document, and are returned to the user's DOM- or SAX-based applications for further processing (see Figure 3.6).

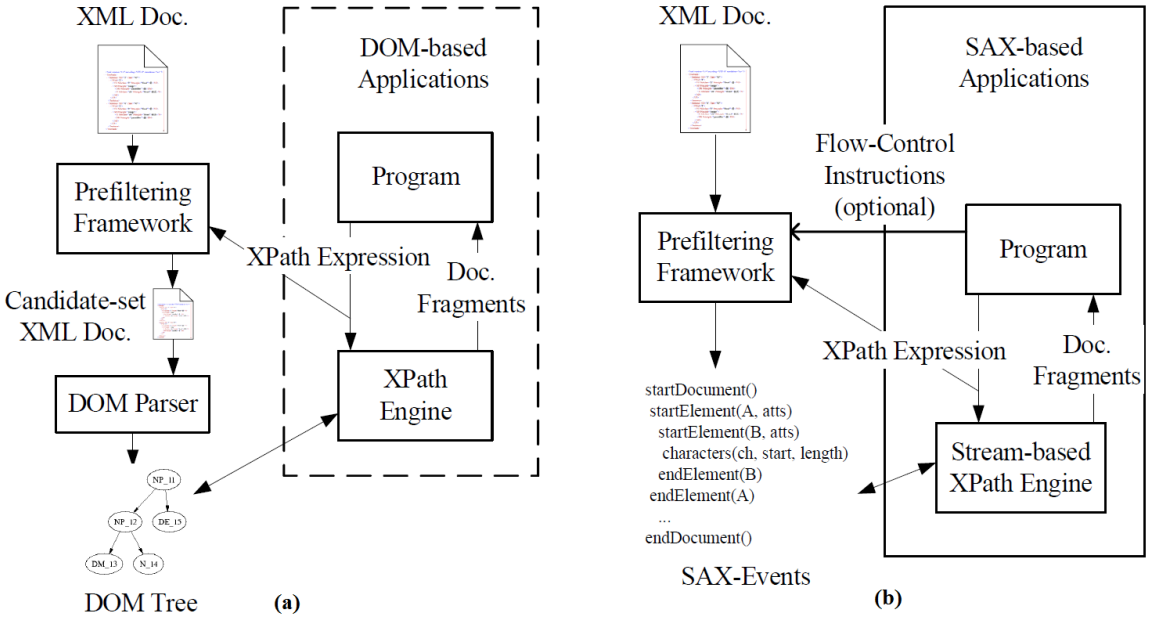


Figure 3.6: Processing model with prefiltering framework.

The main limitation of the prefiltering framework is that it can only be used in the applications that involve query processing. Moreover, as the prefiltering framework need to index the target XML documents and to execute user *XPath* expressions to extract candidate fragments, it is more suitable for the applications that dealing with infrequently updated and large XML documents.

Prefiltering framework consists of five major modules: the *Indexer*, the *Query Simplifier* (*QS*), the *Fast Lightweight Steps-Axes Analyzer* (*FLISA*), the *Fragment Gatherer* (*FG*), and the *Micro XML Streaming Parser* (*MXSP*). The System architecture can be seen in Figure 3.7.

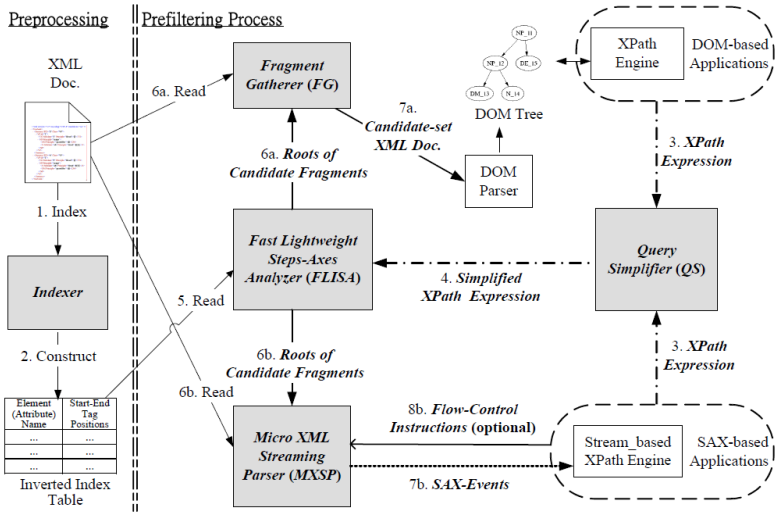


Figure 3.7: The system architecture of the XML prefiltering framework.

3.4.1 Indexer

The Indexer is building up an inverted index table of the XML document. This indexing process only needs to be executed once. After that, the table will be referenced by *FLISA* when evaluating the user queries.

Each record in the table has two fields: *name* and *position list*. The value of the *name* field is either an element name or a string that is concatenated by an attribute name and its value (e.g. SIZE=10). The value of the *position list* is an ordered list; each element of the list is a pair of numbers: (*start-tag position*, *end-tag position*), i.e., the start- and end-byte offsets of an element or an attribute in the XML document. Furthermore, the *position list* is sorted by the *start-tag position*.

3.4.2 Query Simplifier

To reduce the cost of query evaluation, the user XPath expression is simplified by the Query Simplifier module (*QS*). The simplified XPath expression contains fewer steps and has simpler structure compared to those of the original XPath expression. Therefore, the simplified XPath expression can be quickly evaluated.

Authors suggest the following four simplification rules:

(SR1) Omitting internal steps.

(SR2) Omitting branch steps.

(SR3) Omitting wildcard steps.

(SR4) Replacing the parent/child axes with the ancestor/descendant axes.

3.4.3 Fast Lightweight Steps-Axes Analyzer

The Fast Lightweight Steps-Axes Analyzer, *FLISA*, determines the candidate fragments in the XML document by evaluating the simplified XPath expression. Suppose the two steps of a piece of XPath expression are "*u/axis::v*", where *u* and *v* refer to two element names and *axis* ∈ {ancestor, descendant, preceding, following, ancestor-or-self, descendant-or-self, self, attribute}.

No.	*axis*	Evaluation rules
1	Ancestor	*start(v)* < *start(u)* and *end(u)* < *end(v)*
2	Descendant	*start(u)* < *start(v)* and *end(v)* < *end(u)*
3	Preceding	*end(v)* < *start(u)*
4	Following	*end(u)* < *start(v)*

Table 3.2: The equations of evaluating "*u/axis::v*"

An example of how *FLISA* evaluates the XPath expression mentioned above is as follows. First, *u* and *v* are used to select two *position lists*, called *parent_list* and *child_list*, respectively, from the inverted index table. Afterwards, for each *u* in the *parent_list*, find all *v* in the *child_list* such that *Evaluation rule* from Table 3.2 is valid.

Instead of developing another structural query evaluation algorithm to deal with structural queries, the authors slightly modify the *Parent-Child Relationship Filter* (*PCRF*) algorithm developed in their previous work [PCRF algorithm] and apply it to *FLISA*. The

design methodology of *PCRF* is to filter out ineligible candidate fragments from the XML document as soon as possible, and not to spend time evaluating them.

3.4.4 Fragment Gatherer

The candidate fragments *F* determined by *FLISA* can be gathered into one candidate-set XML document *D'* by the *Fragment Gatherer* module *FG* if the user's XPath expression contains only parent, child, ancestor or descendant axes. *D'* consists of two parts: the path information and the candidate fragments.

Generating the candidate fragments *F* is trivial, because the starting- and ending-byte offsets of the root node of a candidate fragment in *D* are known. Generating the path information, however, needs to parse over *D* and to calculate the descendant relationships between the current node *p* in *D* and the root node of *F*. Parsing is started from the root node of *D*. When a start-tag is recognized, its position is used as a key to look up the corresponding end-tag position in the inverted index table. That $p \in D'$ if it contains any candidate fragment as its descendant or it is a candidate fragment; otherwise, the sub-tree rooted by *p* can be ignored by direct moving the file pointer to its end-tag position.

3.4.5 Micro XML Streaming Parser

The Micro XML Streaming Parser, *MXSP*, takes responsibility for transforming the candidate fragments into SAX-events. This procedure is similar to *FG*, but it generates SAX-events instead of a candidate-set XML document.

3.5 ORDPATHs: Insert-Friendly XML Node Labels

[ORDPATH] introduces a hierarchical labeling scheme. ORDPATH labels nodes of an XML tree without requiring a schema. A compressed binary representation of

ORDPATH provides document order by simple byte-by-byte comparison and ancestry relationship equally simply. In addition, the ORDPATH scheme supports insertion of new nodes at arbitrary positions in the XML tree, their ORDPATH values "careted in" between ORDPATHs of sibling nodes, without relabeling any old nodes.

XML data and a tree representing the XML hierarchy are shown in Figures 2.1 and 3.8, respectively, with the corresponding node table shredded from Figure 2.1 shown in Table 3.3. The Node type column of Table 3.3 contains coded values for various node types: 1 for an element, 2 for an attribute, and so on. The Tag column contains coded tags. The VALUE column contains variable-type data that is associated with some nodes.

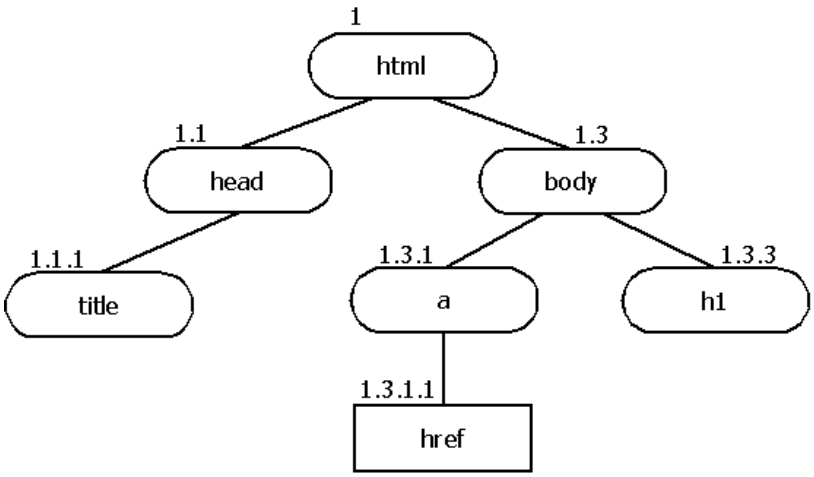


Figure 3.8: XML tree for a document from Figure 2.1

ORDPATH	Tag	Node type	Value
1	1 (html)	1 (Element)	null
1.1	2 (head)	1 (Element)	null
1.1.1	3 (title)	1 (Element)	'My title'
1.3	4 (body)	1 (Element)	null
1.3.1	5 (a)	1 (Element)	'My link'
1.3.1.1	6 (href)	2 (Attribute)	null
1.3.3	7 (h1)	1 (Element)	'My header'

Table 3.3: XML document "shredded" into relational node table

3.5.1 Concepts

Figure 3.9 illustrates successive variable-length L_i/O_i bitstrings of the compressed ORDPATH format.

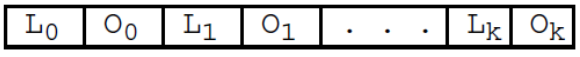


Figure 3.9: Compressed ORDPATH format

In all L_i/O_i component pairs of Figure 3.9, each L_i bitstring specifies the length in bits of the succeeding O_i bitstring. L_i bitstrings are represented using a form of prefix-free encoding, defined in Section 3.5.2, to provide a number of important properties, as follows.

(1) given that we know where an L_i bitstring starts (as we do with L_0), we can identify where it stops;

(2) each L_i bitstring specifies the length in bits of the succeeding O_i bitstring;

(3) from (1) and (2), we see how to parse all ORDPATH bitstrings, left to right, into their L_i/O_i components;

(4) the L_i bitstrings are generated to maintain document order;

(5) L_i/O_i components can specify *negative* ordinals O_i as well as *positive* ones; negative ordinals support multiple inserts of nodes to the left of a set of existing siblings.

3.5.2 L_i/O_i Pair Design

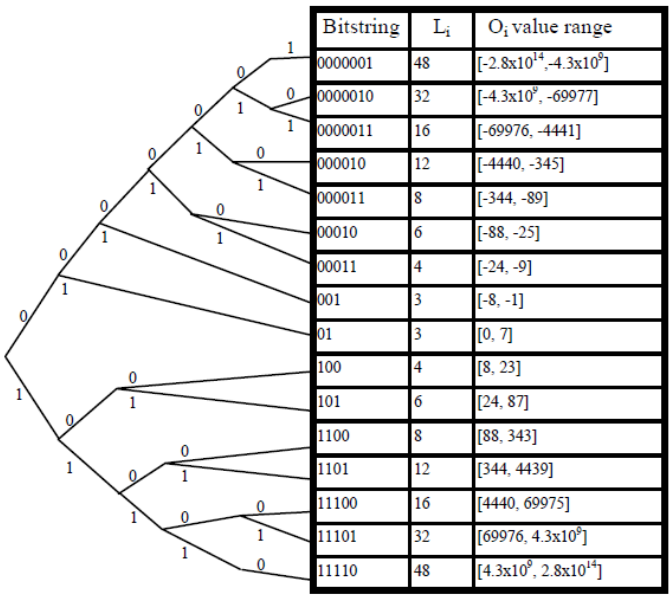
Of the many possible prefix encoding schemes for L_i bitstrings, authors examine two described in Figures 3.10a and 3.10b. In Figure 3.10a, the L_i bitstring 01 identifies a component L_i/O_i encoding with assigned length $L_i = 3$, indicating a 3-bit O_i bitstring. The following O_i bitstrings (000, 001, 010, . . ., 111) represent O_i values of the first eight integers, (0, 1, 2, . . ., 7). Thus 01101 is the bitstring for ORDPATH "5". In the next row in Figure 3.2a, bitstring 100 identifies an encoding with $L_i = 4$ and the 4-bit O_i bitstrings that follow represent the range [8, 23]; in particular, $O_i = 8$ is represented by bitstring 0000, 9 by bitstring 0001,…, up to 23 by bitstring 1111. Similarly, O_i in the range [-8, -1] is associated with the L_i bitstring 001, with -8 represented by the lowest bitstring, 000.

Example 3.1: Using L_i values of Figure 3.10a, we would generate ORDPATH = "1.5.3.-9.11" as follows:

01	001	01	101	01	011	00011	1111	100	0011
L_0=3	O_0=1	L_1=3	O_1=5	L_2=3	O_2=3	L_3=4	O_3=-9	L_4=4	O_4=11

Using the Li values of Figure 3.10b, we would have the following for "1.5.3.-9.11":

01	110	01	10	1	00001	1100	1110	0011
L_0=0 (O_0=1)	L_1=2	O_1=5	L_2=1	O_2=3	L_3=4	O_3=-9	L_4=3	O_4=11



Bitstring	L_i	O_i value range
0000001	48	[$-2.8\text{x}10^{14}$,$-4.3\text{x}10^{9}$]
0000010	32	[$-4.3\text{x}10^{9}$, -69977]
0000011	16	[-69976, -4441]
000010	12	[-4440, -345]
000011	8	[-344, -89]
00010	6	[-88, -25]
00011	4	[-24, -9]
001	3	[-8, -1]
01	3	[0, 7]
100	4	[8, 23]
101	6	[24, 87]
1100	8	[88, 343]
1101	12	[344, 4439]
11100	16	[4440, 69975]
11101	32	[69976, $4.3\text{x}10^{9}$]
11110	48	[$4.3\text{x}10^{9}$, $2.8\text{x}10^{14}$]

a

Bitstring	L_i	O_i value range
000000001	20	[-1118485, -69910]
00000001	16	[-69909, -4374]
0000001	12	[-4373, -278]
000001	8	[-277, -22]
00001	4	[-21, -6]
0001	2	[-5, -2]
001	1	[-1, 0]
01	0	[1, 1]
10	1	[2, 3]
110	2	[4, 7]
1110	4	[8, 23]
11110	8	[24, 279]
111110	12	[280, 4375]
1111110	16	[4376, 69911]
11111110	20	[69912, 1118487]

b

Figure 3.10: Two tables of lengths L_i with O_i ranges represented

3.6 Resume

In chapter 3 we described some of the other techniques used to process XML document and now we try to evaluate them.

A DOM Method to Retrieve Data from a Very Large XML Document

Time

+ Can access widely separately parts of the document at the same time.
+ Building a complete tree may be time consuming.

Memory

+ Do not store the entire XML document into memory.
+ None problem when input XML is large than available memory.
– Need for additional disk capacity for fragment files.

User-friendliness

+ Based on DOM APIs.
– Currently only a small part of DOM API implemented.
– Currently only data read operations.

Hybrid Parallelism for XML SAX Parsing

Time

+ Parser is fast.
+ Using parallelism to improve speed.
– Can access only one event at the time.

Memory

+ Does not store XML into memory.
+ Can access a very large XML documents.

User-friendliness

± Based on SAX APIs with all advantages/disadvantages.

Lazy XML Parsing/Serialization based on Literal and DOM Hybrid Representation

Time

+ Building a partial tree with StAX parser.
+ Speed up deserialization-serialization process.

Memory

– Store entire XML into memory.
– Cannot access a very large XML documents.

User-friendliness

+ Based on DOM APIs.
– Uses only for specific kind of operations such as deserialization-serialization in SOAP messages.

Prefiltering Techniques for Efficient XML Document Processing

Time

+ Uses heuristics to reduce candidate set of an input XML documet.

Memory

+ Suitable for a very large XML documents.

User-friendliness

+ For both DOM and SAX uses.
– Uses only in the applications that involve query processing.
– Uses only for infrequently updated documents.

4 Proposed API

In previous two chapters we discuss and analyse several APIs and techniques for work with XML files. Finally this is the place to propose our own API based on these experiences.

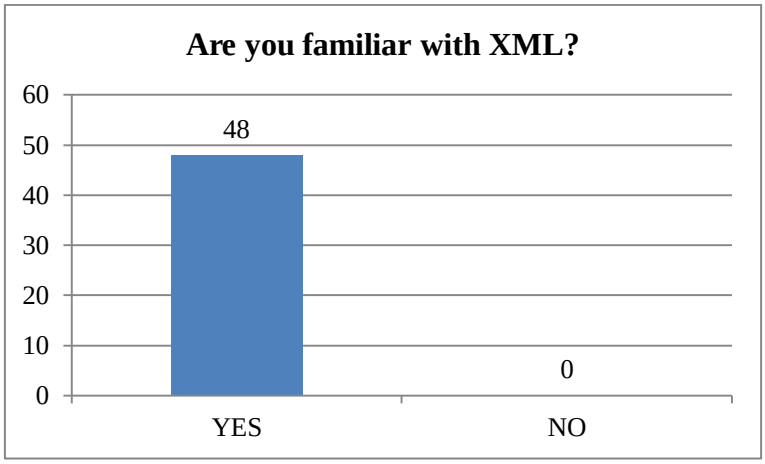
4.1 Motivation

The author of this thesis works, at this time, in organization that is developing information systems for medical purpose. In this area of expertise is very common to communicate between two separated systems through XML files. But it is not only communication. Different medical databases (for example large database [UniProt]) expose their knowledgebase through XML files to specialists or even public. But these XML files may be very large. It is very common that these XML files exceed the size of 10GB. And there is the problem. There are minimal chances to effectively process files at that size. Method mentioned in section 3.1 give us a way to retrieve data from very large xml files by using a general-purpose DOM parser. But this method implements only one function. That has no use for developers, because of missing additional tools or API. And this missing is perfect motivation for our effort.

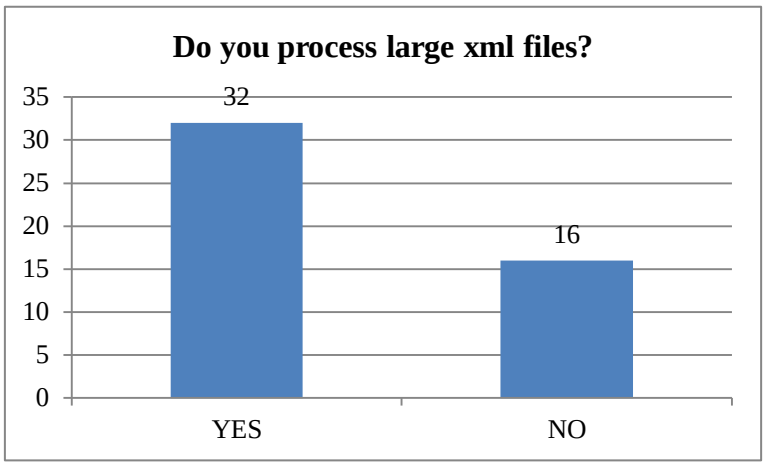
4.2 Question-form Analysis

How motivation suggests, there is one API that solves absence of user friendliness and can use method 3.1. The DOM API (see 2.1) prefect fit, but is this wide known solution really a good one? And is there any probability that this solution will be used? Because of these valid questions we organized small question-form analysis between several database and programme developers. The question-form used for this survey can be found in Appendix B.

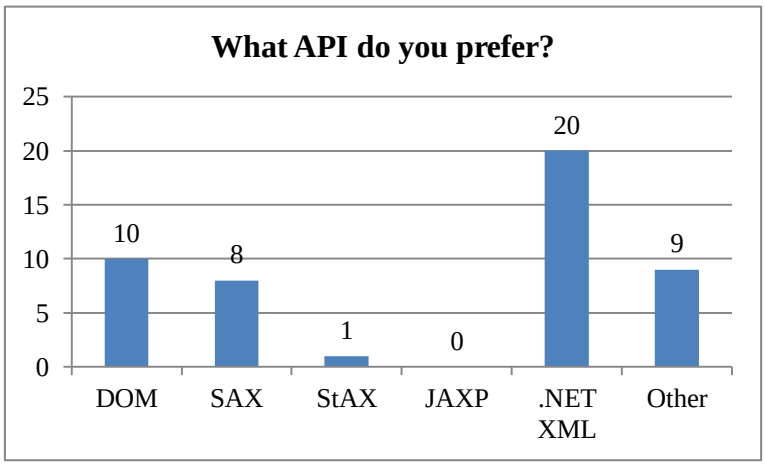
Results is not very predicative, because number of questioned people is only 48 and questioned persons are only from two different companies that are specialized in medical information systems. But these developers often work with large XML files, therefore we mentioned it. Some of the results are shown in Graph 4.1-4.3.



Graph 4.1: Indicates if developers known XML



Graph 4.2: Indicates whether developer process large xml



Graph 4.3: Indicates what API is mainly used

4.3 Simple DOM API for Large XML Files

Simple DOM API for Large XML Files (SDALX) is our name for new proposed API. As the name prompt, new proposed API is DOM based and it is designed for processing large XML files. DOM API is used because of its user friendliness and general knowledge. SDALX implements DOM Level 1 version (see 2.1.3), but really implemented is only data retrieving part of the specification (for more information see chapter 5). For data retrieval operation is used algorithm derived from [KYH DOM] (see 3.1), but is specially modified to fit our proposed API (see 4.3.2).

4.3.1 API interface

SDALX API interface is DOM Level 1 interface. Interface for main Document node is shown in Figure 4.1. For observation of all SDALX API interfaces, please see Appendix C.

```
public interface IDocument : INode
{
        IDocumentType DocumentType { get; }
        DOMImplementation Implementation { get; }
        IElement DocumentElement { get; }

        IElement CreateElement(string tagName);
        IDocumentFragment CreateDocumentFragment();
        IText CreateTextNode(string data);
        IComment CreateComment(string data);
        ICDATASection CreateCDATASection(string data);
        IProcessingInstruction CreateProcessingInstruction(string
        target, string data);
        IAttribute CreateAttribute(string name);
        IEntityReference CreateEntityReference(string name);
        NodeList GetElementsByTagName(string tagname);
}
```

Figure 4.1: Interface for Document node

4.3.2 Algorithm for Data Retrieving

Algorithm 3.1 partitioned an input XML into *n* small documents. Each of the *n* small documents is then modified to meet the well-formedness of the XML document. A data retrieval operation on the original large XML document is then executed sequentially on the small XML tree that is built from each of the modified *n* XML documents, and the results from all the *n* XML trees are combined to generate the final result.

Our proposed algorithm works on similar base, but differently. It also splits an input XML document into *n* smaller documents. Document tree is then built from data retrieval operations on these smaller documents. Data retrieval operations from tree nodes are performed on the document tree or, if this part of document tree is not resolved, on matching small documents. Every node of the document tree obtains list of smaller documents in which is contained. This is guaranteed by *Split* and *Retrieve* algorithm. Each algorithm is explained in further detail below

4.3.3 Algorithm Split

Splitting of an input XML document is executed by one pass of generally-purpose XML pull parser. Formal specification of proposed algorithm is shown below.

Algorithm 8: ***SPLIT(D, n)***

<Input>
1. *D* : An XML document to be partitioned
2. *n* : The expected number of fragments to be generated

<Output>
1. *n'* : The actual number of fragments generated
2. $<D_1\ D_2\ ...D_n>$: A sequence of small XML documents
3. *RootElement* : The root element of an input XML document
4. *XmlDeclaration*: XML declaration of an input XML document

---

```
1  Declare StackOfElement as a stack of Elements;
2  Initialize StackOfElement;
3  Declare XmlReader as a general-purpose Pull parser.
4  Declare XmlWriter as a general-purpose Xml writer.
5  XmlDeclaration := the empty XmlDeclaration;
6  RootElement := the empty Element.
7  FilePartLength :=D.Length / n;
8  ElementIdentification := 0;
9  ResolvedElement := the empty Element;
```

```
Initialize XmlWriter to first split document
n':=1;
While (XmlReader.Read) do
    If(XmlReader.Node is XmlDeclaratin )
        Initialize XmlDeclaration from XmlReader.Node
        XmlWriter.WriteXmlDeclaration(XmlReader.Node)
    EndIf
    If (XmlReader.Node is EndElement)
        Pop the last item from StackOfElement;
        XmlWriter.WriteEndElement()
    EndIf
    If (XmlReader.Node is Element )
        Increase ElementIdentification;
        Initialize ResolvedElement from XmlReader.Node;
        If (RootElement is empty)
            RootElement is ResolvedElement;
        EndIf
        XmlWriter.WriteStartElement(ResolvedElement);
        Add an identification attribute with value of ElementIdentification to ResolvedElement
        If (ResolvedElement is an empty element)
            XmlWriter.WriteElementAttributes(ResolvedElement);
            XmlWriter.WriteEndElement(ResolvedElement);
        Else
            Push ResolvedElement into StackOfElement;
            If (Length of file created by XmlWriter > FilePartLength)
                XmlWriter.WriteElementAttribute(CutElement);
                Increase n';
                Initialize XmlWriter2 to n' split document;
                XmlWriter2.WriteXmlDeclaration(XmlDeclaration);
                For each Element in StackOfElement in FIFO order
                    XmlWriter.WriteEndElement();
                    XmlWriter2.WriteStartElement(Element);
                    XmlWriter2. WriteElementAttributes(Element);
                    If (Element is ResolvedElement)
                        XmlWriter2.WriteElementAttribute(CutElement);
                    Else
                        XmlWriter2.WriteElementAttribute(DummyElement);
                    EndIf
                EndFor
                XmlWriter.CloseFile();
                XmlWriter := XmlWriter2;
            EndIf
        EndIf
    EndIf
    If (XmlReader.Node is not (Element, EndElement, XmlDeclaration))
        XmlWriter.Write(XmlReader.Node);
    EndIf
End while
XmlWriter.CloseFile();
Return n', <D1 D2 ... Dn'>, RootElement and XmlDeclaration;
```

On the beginning we compute length for split document. Our algorithm next simply iterates the input XML document through pull parser; sequentially read content is written to smaller document by generally-purpose xml writer. When a *StartElement* is pulled out evaluations begins. To each pulled element an identification attribute is added. If element is empty, nothing happens. Element is only written by xml writer to smaller document. If it

is not empty, then besides of writing by xml writer; to stack of element is pushed. If it is not empty and size of an xml document, written by xml writer, exceeds computed length for splitting then a split operation is performed. Split operation consist of initializing of second xml writer, that write xml declaration and start tags of all element stored in stack in FIFO order. To each element except last one (in FIFO order) a ***dummy attribute*** is added. *Dummy attribute* differentiate original elements from elements, which are generated by this algorithm and which are not last in FIFO order in stack. To this last element a ***cut attribute*** is added. *Cut attribute* only mark elements where the document split occurs. Finally the first xml writer writes all end tags from elements stored in stack in FILO order and closes its document. Then second writer becomes first and loop continues to the end of input document.

4.3.4 Algorithm Retrieve

Formal specification of our proposed algorithm for data retrieval operation is shown below.

Algorithm 9: ***RETRIEVE(D, E, P)***

<Input>
1. *D* : SDALX document node
2. E : An SDALX element of *D*
3. *P*: A condition that can be satisfied by an element of *D*

<Output>
1. N_r: A node list with results.

```
1   Declare N_r as an empty SDALX node list;
2   Declare DP as a general- purpose DOM parser;
3   Declare HashElementList as an empty fast list of resolved SDALX elements
4   For each D_i from E.InFiles
5       T_i := Build an XML tree from D_i by using DP;
6       e := Find corresponding element of E using DP and identification attribute;
7       N_i := Execute GET(T_i, e, P) by using DP;
8       For each node e of N_i
9           Create SDALX node E from e
10          If (E is an element)
11              If (e has CutAttribute or DummyAttribute)
12                  If (HashElementList not contains E)
13                      Add E into HashElementList;
14                      Add E into N_r;
15                  EndIf
16              Else
17                  Add E into HashElementList;
```

```
18                  Add E into N_r;
19              End if
20          Else
21              Add E into N_r;
22          End if
23      End for
24  End for
25  Return N_r;
```

Data retrieving algorithm is simple. On the input algorithm gets document node that contains list of all small documents and list of resolved elements; gets also document element from which retrieving begin and condition that must be satisfied. In the contrast of KYH DOM a condition *P*, given on the input, solves also form of output. For better understanding we could project condition *P* as an XPath query. An element *E* also contains list of all small documents where it belongs; in algorithm it is given by *InFiles* statement.

Algorithm browses all documents which contain element *E*. Each document is probed by generally-purpose DOM parser to find corresponding element *e* with the same identification attribute as element *E* has. If the element is found, then get statement on this element is executed. Finally for every result node a corresponding SDALX one is created. If it is an element, then node *e* is tested if there is a cut or dummy attribute. If there is one, then an element could be resolved before, therefore *HashElementList* that contains all resolved elements by this query must be probed. Only unresolved element or other node is added to node list.

5 Implementation

In this chapter we describe our implementation of proposed API and splitting and retrieving algorithm from Chapter 4. We decided to implement our API in .NET framework version 4.0. That because it contains very elaborate interface for XML processing and our effort is to offer similar DOM functions like *XmlDocument* interface within .NET XML library even for a very large XML documents.

Our API is DOM Level 1 based, therefore it must implement basic interfaces from DOM Level 1 specification. These interfaces can be found in Appendix C. Please be noted that some methods from these interfaces are not implemented, respectively they throw a *NotImplementedException*. Implemented are basic retrieval functions from all interfaces. Basic implemented class like are furthermore explained below. For detailed information about all classes please see manual.

5.1 Node Class

Node class is a basic class, because *Node* is a basic tree element. Implemented are all properties. See below.

Public properties	Description
NodeName	Gets the qualified name of the node.
NodeValue	Gets or sets the value of the node.
NodeType	Gets the type of the current node.
ReadOnly	Gets a value indicating whether the node is read-only.
ParentNode	Gets the parent of this node (for nodes that can have parents).
ChildNodes	Gets all the child nodes of the node.
FirstChild	Gets the first child of the node.
LastChild	Gets the last child of the node.
PreviousSibling	Gets the last child of the node.
NextSibling	Gets the node immediately following this node.
Attributes	Gets an *AttributeCollection* containing the attributes of this node.
OwnerDocument	Gets the *LargeXmlDocument* to which this node belongs.

Table 5.1: Public properties of Node class

Almost all methods from *INode* interfaces that *Node* class must implement cause data changes, therefore are not implemented, they are only exposed. If not implemented method is accessed, then *NotImplementedException* is thrown.

Please noted, that *Node* class internally stored list of xml files in which is located. Any retrieving action to subtree of the node causes that only files from list are probed by DOM parser.

5.2 LargeXmlDocument Class

LargeXmlDocument class implements *IDocument* interface and it is class that represent an XML document. *LargeXmlDocument* inherits from *Node* class, therefore a lot of properties and methods are the same. List of implemented properties and methods from *IDocument* interface is shown in Table 5.2.

Public properties	Description
DocumentElement	Gets root element of the document.
DocumentType	Gets the node containing the DOCTYPE declaration.

Table 5.2: Public properties from IDocument interface

List of other public properties is shown in Table 5.3.

Public properties	Description
Config	Gets or sets configuration for partition process.
IsXmlLoaded	Indicates whether an xml file is loaded or not.
NumberOfFragments	Number of fragments for current XML file.

Table 5.3: Public properties of LargeXmlDocument

List of implemented methods:

GetElementsByTagName(tagname)

Return a list of all descendant elements that match the specified *tagname*. This method implements our Retrieving algorithm.

SelectNodes(xpath)

Return a list of nodes matching the *xpath* expression.

This method is not mentioned in *IDocument* interface, but was added to improve query capabilities of our API.

This method implements our Retrieving algorithm.

Load(filename)

Load the XML document from the specified address.

This method is not mentioned in *IDocument* interface, but is very important, because performs load of XML Document using our *Split* algorithm.

Close()

Close currently loaded XML and release resources.

The main task of this method is to clean up all resources and delete all smaller XML files generated by *Split* algorithm in *Load* method.

LargeXmlDocument class also internally store a set of all resolved elements. This set is very helpful for resolving connections between separated nodes in our partial DOM tree.

5.3 Element Class

Element class implements *IElement* interface and it is class that represent an XML element. *Element* inherits from *Node* class. *IElement* interface adds only new methods and list of these implemented methods is shown below.

List of implemented methods:

GetAttribute(name)

Return the value for the attribute with the specified name.

SetAttribute(name, value)

Set the value of the attribute with the specified name.

RemoveAttribute(name)

Remove an attribute by name.

GetAttributeNode(name)

Return the *Attribute* with the specified name.

SetAttributeNode(newAttr)

Add the specified Attribute.

RemoveAttributeNode(oldAttr)

Remove the specified Attribute.

GetElementsByTagName(name)

Return a list of all descendant elements that match the specified *name*.

This method implements our Retrieving algorithm.

5.4 Attribute Class

Attribute class implements *IAttribute* interface and it is class that represent an attribute of the XML element. *Attribute* inherits from *Node* class, but is not a tree node in the context of DOM tree. List of implemented properties of *IAttribute* interface is shown in Table 5.4.

Public properties	Description
Specified	Gets a value indicating whether the attribute value was explicitly set.
OwnerElement	Gets the Element to which this attribute belongs.
HasDefaultValue	Indicates whether attribute has a default value or not.

Table 5.4: Public properties from IAttribute interface

5.5 Config Class

Config class is a simple class, but with great importance. By handling this class to *LargeXmlDocument* class by *LargeXmlDocument.Config* property we can alter settings of *Split* methods. List of implemented properties is shown in Table 5.5.

Public properties	Description
AutoComputeNumberOfFragmets	Indicates whether an automatical computing for number of fragments may be used.
EstimatedNumberOfFragments	Estimated number of fragments for current XML file.
AutoSelectPathForFileFragments	Indicates whether an automatical path for file fragments may be used.
PathForFileFragments	Path for file fragments.

Table 5.5: Public properties of Config class

5.6 XmlSplitter Class

XmlSplitter class is internal (not visible for user), but also with great importance. Contains only one method *Split(xmlFilename, doc)*; but this methods implements our *Split* algorithm on input XML document referenced by *xmlFilename*; splitting configuration is stored in *LargeXmlDocument* parameter *doc*.

6 Experiments

In this chapter we evaluate the performance of the proposed SDALX API, that implementation is described in previous chapter, to process large XML document.

Because of outer similarity of our parsing methods to KYH DOM we propose to conduct similar testing scenarios like authors of KYH DOM. That is because is very hard to choose another testing scenario with appropriate set of queries for very large XML documents. So we made two test scenarios; one process various sized XML documents generated by [xmlgen] for performance evaluation only, and the other process very large XML documents which are used in medical realm and are also downloadable from the Internet without any feeds or permissions required.

The hardware platform of this experiment was a personal computer with an Intel® Core™2 DUO CPU (2.67 GHz, E7300) with 6 GB of virtual memory (4 GB of physical memory plus 2 GB of swap space) and which ran Microsoft® Windows Vista™ Business as its operating system.

6.1 Testing Scenario A

6.1.1 Test Data

We concluded (like authors of KYH DOM) to use automatically generated XML data from the XML generator xmlgen. We generated a set of XML documents from 100 MB to 1 GB in increments of 100 MB. Structure of generated data can be seen in Figure 6.1 (figure is passed from KYH DOM).

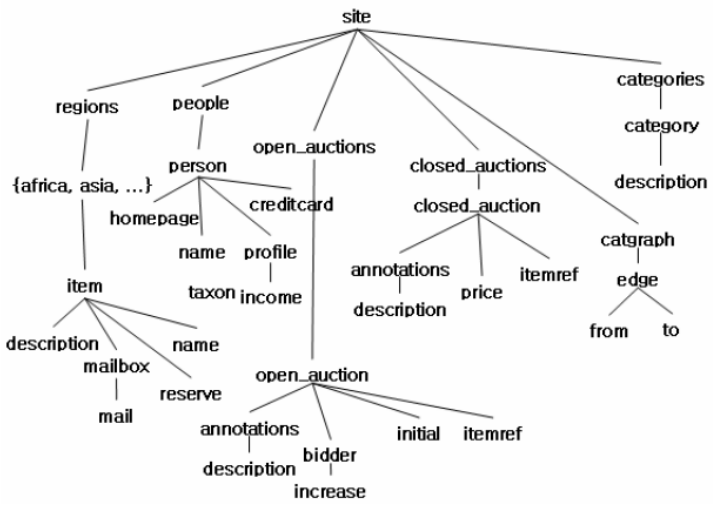


Figure 6.1: Structure of a document generated by xmlgen

We evaluated the processing time required to execute GET operations, which are expressed with .NET XML DOM API and SDALX DOM API, on each of generated XML documents. Twenty queries from [XMark] were chosen as a representative of these GET operations because are used widely in many research papers for benchmark performance on the files generated by xmlgen. List of these queries is given below.

Q1. Return the name of the person with ID 'person0'.

Q2. Return the initial increases of all open auctions.

Q3. Return the IDs of all open auctions whose current increase is at least twice as high as the initial increase.

Q4. List the reserves of those open auctions where a certain person issued a bid before another person.

Q5. How many sold items cost more than 40?

Q6. How many items are listed on all continents?

Q7. How many pieces of prose are in our database?

Q8. List the names of persons and the number of items they bought.

Q9. List the names of persons and the names of the items they bought in Europe.

Q10. List all persons according to their interest; use French markup in the result.

Q11. For each person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.

Q12. For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.

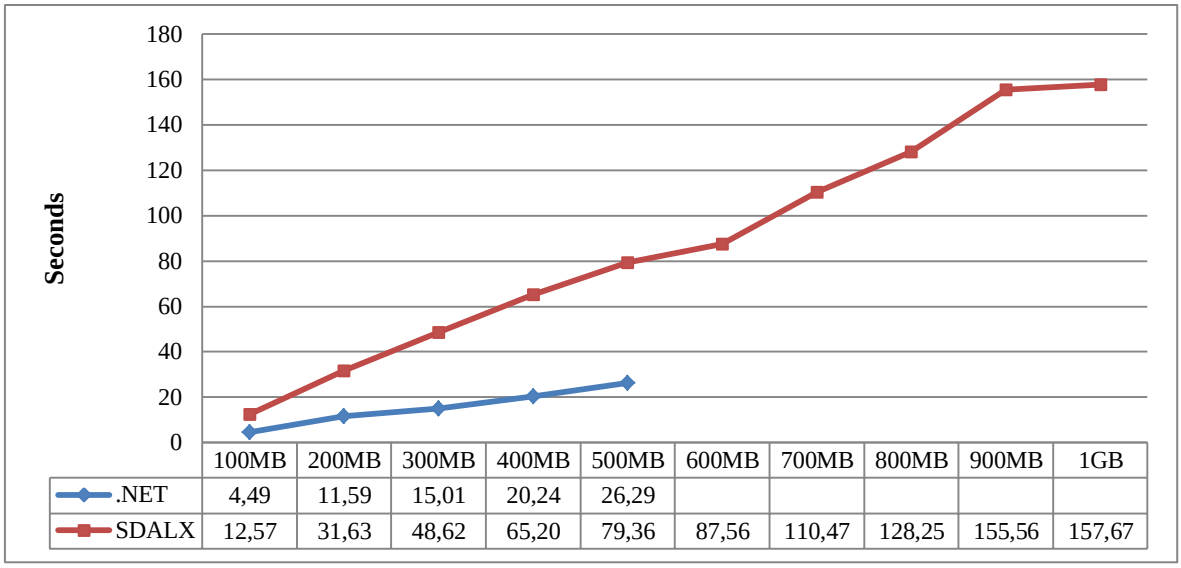
Q13. List the names of items registered in Australia along with their descriptions.

Q14. Return the names of all items whose description contains the word 'gold'.
Q15. Print the keywords in emphasis in annotations of closed auctions.
Q16. Return the IDs of those auctions that have one or more keywords in emphasis.
Q17. Which persons don't have a homepage?
Q18. Convert the currency of the reserve of all open auctions to another currency.
Q19. Give an alphabetically ordered list of all items along with their location.
Q20. Group customers by their income and output the cardinality of each group.

Queries are used for an XQuery processor and any XQuery using DOM API to access XML documents executes a number of GET operations for finding nodes, which are specified in any given query. Therefore, in the viewpoint of checking the performance of GET operations, there are no critical differences in these queries. So we (like authors of KYH DOM) selected two queries that need relatively short execution time: queries Q1 and Q14.

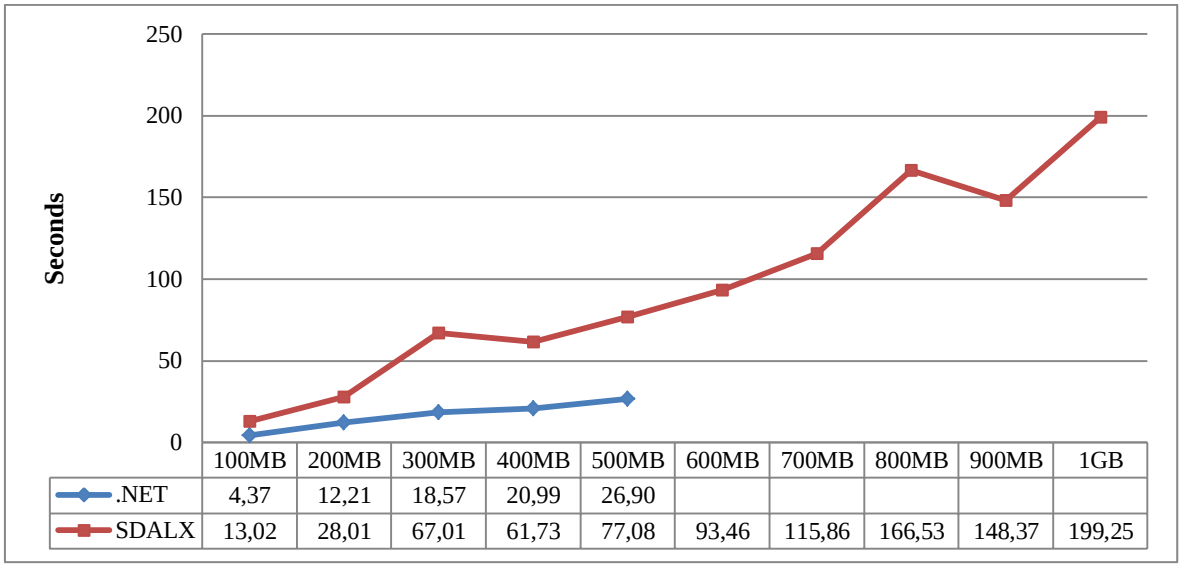
6.1.2 Test Results

Figures 6.1 and 6.2 show the execution time of Q1 and Q14 on 10 generated XML documents correspondingly. We first computed the execution time of .NET DOM library and then we computed the execution plan of SDALX library for every XML document. On the end of each test the garbage collector was called to forces an immediate garbage collection of all generations, to minimalize calling of garbage algorithm during tests. We performed every experiment on each XML document 5 times to get the average time as an execution time. In these figures, the time required for splitting algorithm is included in the total time of query execution.



	100MB	200MB	300MB	400MB	500MB	600MB	700MB	800MB	900MB	1GB
.NET	4,49	11,59	15,01	20,24	26,29					
SDALX	12,57	31,63	48,62	65,20	79,36	87,56	110,47	128,25	155,56	157,67

Figure 6.2: Execution of Q1 on various-sized XML Documents from 100 MB to 1 GB



	100MB	200MB	300MB	400MB	500MB	600MB	700MB	800MB	900MB	1GB
.NET	4,37	12,21	18,57	20,99	26,90					
SDALX	13,02	28,01	67,01	61,73	77,08	93,46	115,86	166,53	148,37	199,25

Figure 6.3: Execution of Q14 on various-sized XML Documents from 100 MB to 1 GB

From both figures is clear that .NET DOM library is faster up to 500MB and suddenly there are no data. That is because of the shortage of memory for .NET DOM parser on files greater than 500MB. On the other hand SDALX library works fine and execution time grows linearly with the size of an input XML file.

SDALX *Split* algorithm was set to automatically compute number of fragments. That means that an input XML document is split always after approximately 25MB of data. Number of fragments generated on each xml files is given in Table 6.1.

XML document size	Number of fragments
100 MB	5
200 MB	12
300 MB	17
400 MB	23
500 MB	28
600 MB	34
700 MB	39
800 MB	45
900 MB	50
1 GB	57

Table 6.1: Number of generated fragments by Split algorithm

The results of these experiments show that the implemented library of proposed API can be used for processing large XML files on the one single general-purpose personal computer. On smaller files is better to use standard .NET DOM library.

But one disadvantage the new proposed API has; for processing very large XML files is required approximately 1,5 times more space, than the size of an input large XML is.

6.2 Testing Scenario B

6.2.1 Test Data

Second testing scenario is from real life. Like authors of KYH DOM we choose a large XML document containing Protein Sequence Database from UniProt. But contrary to KYH DOM we choose *uniref50.xml* to evaluate our method. DTD of chosen document is given in Figure 6.4.

```
<!ELEMENT UniRef50 (entry+)>
<!ATTLIST UniRef50 xmlns CDATA #FIXED "http://uniprot.org/uniref"
                   xmlns:xsi CDATA #IMPLIED
                   xsi:schemaLocation CDATA #IMPLIED
                   releaseDate    CDATA #IMPLIED
                   version        CDATA #IMPLIED>

<!-- entry: UniRef50 entry -->
<!ELEMENT entry (name,property*,representativeMember,member*)>
<!ATTLIST entry  id             ID    #REQUIRED
                 updated        CDATA #IMPLIED
>

<!-- name: UniRef50 cluster name derived from representative -->
<!-- UniRef100 entry  -->
<!ELEMENT name  (#PCDATA)>

<!-- representativeMember: information for representative -->
<!-- UniRef100 entry  -->
<!ELEMENT representativeMember (dbReference,sequence)>

<!-- memberList: members of UniRef50 cluster other than representative -->
<!ELEMENT member (dbReference)>

<!-- dbReference: cross-reference to member UniRef100 entries  -->
<!-- of the UniRef50 cluster -->
<!ELEMENT dbReference (property*)>
<!ATTLIST dbReference
    type CDATA #REQUIRED
    id   CDATA #REQUIRED
>

<!-- property: properties of cross-references -->
<!ELEMENT property EMPTY>
<!ATTLIST property
    type CDATA #REQUIRED
    value CDATA #REQUIRED
>

<!ELEMENT sequence (#PCDATA ) >
<!ATTLIST sequence
    length CDATA #IMPLIED
    checksum CDATA #IMPLIED
>
```

Figure 6.4: DTD of uniref50.xml

From this document, that is a simple aggregation of protein sequences, we perform a simple query: QU: "*Return all gene names.*".

6.2.2 Test Results

Figure 6.5 show the split time, execution time and the total time of QU on a very large XML document *uniref50.xml*. There is no figure for .NET DOM library because of the shortage of memory for .NET DOM parser. We performed experiment 5 times only for SDALX library.

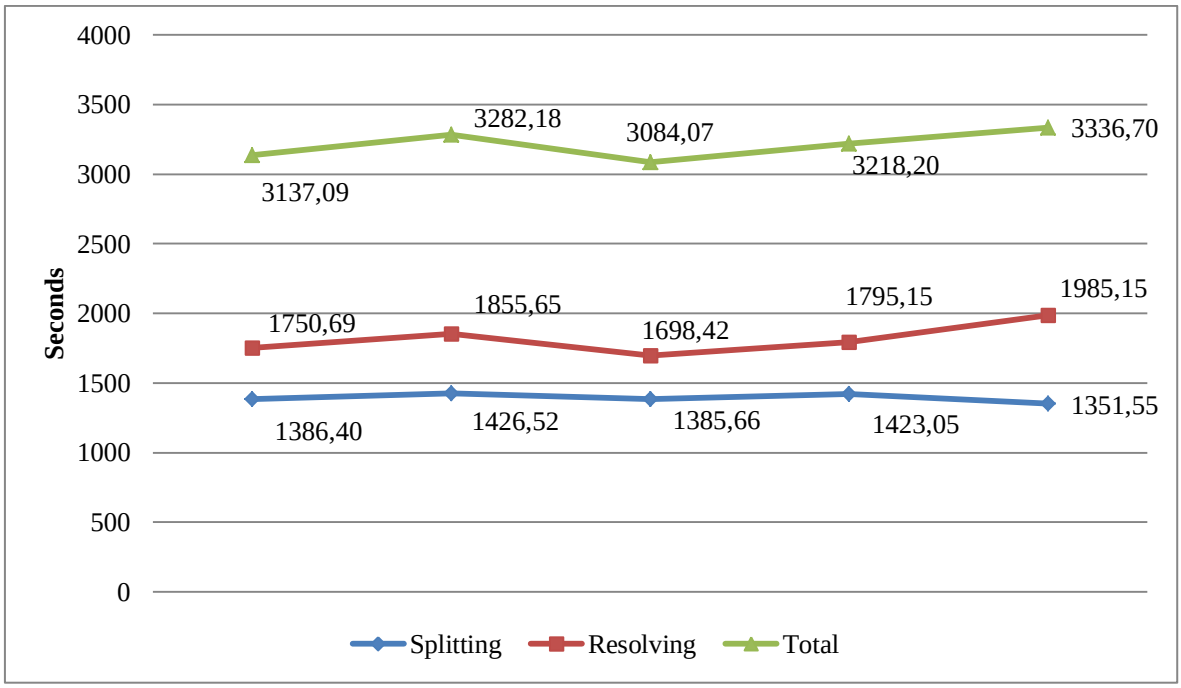


Figure 6.5: Retrieving gene names from uniref50.xml

Number of fragments generated by our *Split* algorithm was 869 for *uniref50.xml*. Figure also show that proposed SDALX library may be used even for XML documents larger than 15GB. There is only one limiting condition; the size of the virtual memory. If the number of resulted nodes is greater than the capacity of virtual memory, then even our library does not work and shortage of memory occurs.

7 Conclusions

The aim of this thesis was to propose a new API for XML data on the basis of previous analysis the most known APIs for processing XML documents.

In this thesis, we presented and partial implemented a new DOM like API with purpose to process very large XML documents. This framework was inspired by .NET XML library and its aim was to provide to user similar possibilities for work with very large XML documents which .NET XML library provides. Our framework used the idea of splitting input large XML documents into several small documents. User tasks are then performed on these small documents and results from these small documents are merged and returned to user; consistently are merged results used in building internal partial DOM tree.

The idea of splitting an input XML document comes from KYH DOM mentioned in chapter 3.1. There are also presented algorithm that performs partitioning input large XML document, after this is finished the padding algorithm, that guarantees well-formedness of small XML documents, occurs; and finally retrieving algorithm, that returns results to user, is performed. We used this idea in our own algorithm presented in chapter 4.3.

Experimental results indicates that our framework is suitable for work with large XML document; especially when an XML document is very large and various operations for finding elements are to be executed on this XML document frequently. Our method for splitting input XML and querying these splits documents, presented in this thesis, holds the execution time and memory using in the level of usefulness.

7.1 Future Work

As already mentioned this work implements only part of DOM APIs; data retrieval part to be precisely. Extending implementation for methods supporting INSERT, DELETE, and UPDATE operations on very large XML is the next logical step.

Bibliography

[HTML 4.0] D. Raggett, A. Le Hors, I. Jacobs: *HTML 4.0 Specification*, W3C Recommendation, April 1998. http://www.w3.org/TR/1998/REC-html40-19980424

[HTML 4.01] D. Raggett, A. L. Hors, I. Jacobs: *HTML 4.01 Specification*, W3C Recommendation, December 1999. http://www.w3.org/TR/html401/

[JAXP] *JAXP Project*, https://jaxp.dev.java.net/

[Java] *Java programming language*, http://www.java.com

[KYH DOM] S. M. Kim, S. I. Yoo, E. Hong, T. G. Kim, I. K. Kim: *A Document Object Modeling Method to Retrieve Data from a Very Large XML Document*, In *Proceedings of the 2007 ACM Symposium on Document engineering*, August 2007.

[Meta-DFA] Y. Pan, Y. Zhang, K. Chiu, W. Lu: *Parallel XML Parsing Using Meta-DFAs*, In *3rd IEEE International Conference on e-Science and Grid Computing*, December 2007.

[MS: DHTML] About the DHTML Object Model, In *Microsoft Developer Network*. http://msdn.microsoft.com/en-us/library/ms533022(v=vs.85).aspx

[OASIS: UDDI] Tom Bellwood: *UDDI Version 2.04 API Specification*, UDDI Committee Specification, July 2002. http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm

[ORDPATH] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury: *ORDPATHs: insert-friendly XML node labels*. In *ACM SIGMOD International Conference on Management of Data*, June 2004. http://portal.acm.org/citation.cfm?id=1007686#references

[PCRF algorithm] C. H. Huang, T. R. Chuang, and H. M. Lee: *Fast Structural Query with Application to Chinese Treebank Sentence Retrieval*, In *ACM Symposium on Document Engineering*, October 2004.

[PZC SAX] Y. Pan, Y. Zhang, K. Chiu: *Hybrid Parallelism for XML SAX Parsing*, In *2008 IEEE International Conference on Web Services*, 2008.

[SAX] *SAX Project*, http://www.saxproject.org/

[StAX] Java Community Process: *Streaming API For XML*, JSR-173 Specification, October 2003. http://jcp.org/en/jsr/detail?id=173&showPrint

[StAX: Cursor API] Java Community Process: *Streaming API For XML*, JSR-173 Specification, pages 44 – 48, October 2003.

[StAX: Event Iterator API] Java Community Process: *Streaming API For XML*, JSR-173 Specification, pages 50 – 56, October 2003.

[TaT DOM] T. Takase, K. Tajima: *Lazy XML Parsing/Serialization based on Literal and DOM Hybrid Representation*, In *2008 IEEE International Conference on Web Services*, 2008.

[UniProt] *UniProt*, http://www.uniprot.org/

[W3C] *World Wide Web Consotium*, http://www.w3.org/

[W3C: CSS] B. Bos, T. Çelik, I. Hickson, H. W. Lie: *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*, W3C Recommendation, June 2011. http://www.w3.org/TR/2011/REC-CSS2-20110607/

[W3C: DOM] W3C DOM Interest Group: *Document Object Model (DOM)*, W3C Recommendations, January 2005. http://www.w3.org/DOM/

[W3C: DOM Activity] Various Working Groups within the W3C: *Document Object Model Activity Statement*, W3C Activity Statement, January 2008. http://www.w3.org/DOM/Activity/

[W3C: DOM Level 1] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood: *Document Object Model (DOM) Level 1 Specification*, W3C Recommendation, October 1998. http://www.w3.org/TR/REC-DOM-Level-1/

[W3C: DOM Level 1 Core] M. Champion, S. Byrne, G. Nicol, L. Wood: *Document Object Model (Core) Level 1*, First bulleted list of [W3C: DOM Level 1], http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html

[W3C: DOM Level 1 HTML] M. Champion, V. Apparao, S. Isaacs, C. Wilson, I. Jacobs: *Document Object Model (HTML) Level 1*, Second bulleted list of [W3C: DOM Level 1], http://www.w3.org/TR/REC-DOM-Level-1/level-one-html.html

[W3C: DOM Level 2 Core] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne: *Document Object Model (DOM) Level 2 Core Specification*, W3C Recommendation, November 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/

[W3C: DOM Level 2 Events] Tom Pixley: *Document Object Model (DOM) Level 2 Events Specification*, W3C Recommendation, November 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113/

[W3C: DOM Level 2 HTML] J. Stenback, P. L. Hégaret, A. L. Hors: *Document Object Model (DOM) Level 2 HTML Specification*, W3C Recommendation, November 2000. http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/

[W3C: DOM Level 2 Style] C. Wilson, P. L. Hégaret, V. Apparao: *Document Object Model (DOM) Level 2 Style Specification*, W3C Recommendation, November 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Style-20001113/

[W3C: DOM Level 2 Traversal and Range] J. Kesselman, J. Robie, M. Champion, P. Sharpe, V. Apparao, L. Wood: *Document Object Model (DOM) Level 2 Traversal and Range Specification*, W3C Recommendation, November 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113/

[W3C: DOM Level 2 Views] A. L. Hors, L. Cable: *Document Object Model (DOM) Level 2 Views Specification*, W3C Recommendation, November 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Views-20001113/

[W3C: DOM Level 3 Abstract Schemas] B. Chang, E. Litani, J. Kesselman, R. Rahman: *Document Object Model (DOM) Level 3 Abstract Schemas Specification*, W3C Note, July 2002. http://www.w3.org/TR/2002/NOTE-DOM-Level-3-AS-20020725/

[W3C: DOM Level 3 Core] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne: *Document Object Model (DOM) Level 3 Core Specification*, W3C Recommendation, April 2004. http://www.w3.org/TR/DOM-Level-3-Core/

[W3C: DOM Level 3 Events] P. L. Hégaret, T. Pixley: *Document Object Model (DOM) Level 3 Events Specification*, W3C Working Group Note, November 2003. http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107/

[W3C: DOM Level 3 Load and Save] J. Stenback, A. Heninger: *Document Object Model (DOM) Level 3 Load and Save Specification*, W3C Recommendation, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-LS-20040407/

[W3C: DOM Level 3 Validation] B. Chang, J. Kesselman, R. Rahman: *Document Object Model (DOM) Level 3 Validation Specification*, W3C Recommendation, January 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Val-20040127/

[W3C: DOM Level 3 Views and Formatting] Ray Whitmer: *Document Object Model (DOM) Level 3 Views and Formatting Specification*, W3C Working Group Note, February 2004. http://www.w3.org/TR/2004/NOTE-DOM-Level-3-Views-20040226/

[W3C: DOM Level 3 XPath] Ray Whitmer: *Document Object Model (DOM) Level 3 XPath Specification*, W3C Working Group Note, February 2004. http://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226/

[WRC: SOAP 1.1] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer: *Simple Object Access Protocol (SOAP) 1.1*, W3C Note, May 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[W3C: WSDL 1.1] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana: *Web Services Description Language (WSDL) 1.1*, W3C Note, March 2001. http://www.w3.org/TR/wsdl

[W3C: XML] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau*: Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/

[W3C: XML Base] Jonathan Marsh: *XML Base*, W3C Recommendation, June 2001. http://www.w3.org/TR/2001/REC-xmlbase-20010627/

[W3C: XML Information Set] J. Cowan, R. Tobin: XML *Information Set (Second Edition)*, W3C Recommendation, February 2004. http://www.w3.org/TR/2004/REC-xml-infoset-20040204/

[W3C: XML Namespaces] T. Bray, D. Hollander, A. Layman, R. Tobin, H. S. Thompson: *Namespaces in XML 1.0 (Third Edition)*, W3C Recommendation, December 2009. http://www.w3.org/TR/REC-xml-names/

[W3C: XML Schema - Datatypes] P. V. Biron, A. Malhotra: *XML Schema Part 2: Datatypes Second Edition*, W3C Recommendation, October 2004. http://www.w3.org/TR/xmlschema-1/

[W3C: XML Schema - Structures] H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn: *XML Schema Part 1: Structures Second Edition*, W3C Recommendation, October 2004. http://www.w3.org/TR/xmlschema-1/

[W3C: XPath 1.0] J. Clark, S. DeRose: *XML Path Language (XPath)*, W3C Recommendation, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116/

[W3C: XSLT] James Clark: *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation, November 1999. http://www.w3.org/TR/xslt

[Wikipedia: SAX] *Simple API for XML*, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Simple_API_for_XML

[XHTML 1.0] S. Pemberton, D. Austin, J. Axelsson, T. Çelik, D. Dominiak, H. Elenbaas, B. Epperson, M. Ishikawa, S. Matsui, S. McCarron, A. Navarro, S. Peruvemba, R. Relyea, S. Schnitzenbaumer, P. Stark: *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)*, W3C Recommendation, January 2000. http://www.w3.org/TR/xhtml1/

[XML in .NET] Aaron Skonnard: *XML in .NET: .NET Framework XML Classes and C# Offer Simple, Scalable Data Manipulation*, MSDN Magazine, January 2001. http://msdn.microsoft.com/en-us/magazine/cc302158.aspx

[XML Prefiltering] C. H. Huang, T. R. Chuang, H. M. Lee: *Prefiltering Techniques for Efficient XML Document Processing*, In *ACM Symposium on Document Engineering*, November 2005.

[XMark] *XMark Benchmark Queries*, http://www.ins.cwi.nl/projects/xmark/Assets/xmlquery.txt

[xmlgen] *xmlgen - The Benchmark Data Generator*, http://www.ins.cwi.nl/projects/xmark/generator.html

[XQuery] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon: *XQuery 1.0: An XML Query Language (Second Edition)*, W3C Recommendation, December 2010. http://www.w3.org/TR/2010/REC-xquery-20101214/

Appendix A

Content of DVD

The DVD attached to this thesis has the following structure:

- `content.txt` - A file with this text.
- `text/` - A PDF version of the thesis.
- `sdalx-lib/` - Compiled SDALX library for DOM parsing of very large XML
- `sdalx/` - Source codes of the SDALX library for DOM parsing of very large XML.
- `test/` - A directory containing sets of test data and test results.

Appendix B

Question form

1. Are you familiar with XML?

   YES NO

2. What kind of work with XML do you prefer?

   Work with tree. Work with events. Other.

3. What API do you prefer and why?

   - DOM ……
   - SAX ……
   - StAX ……
   - JAXP ……
   - .NET XML ……
   - Other ……

4. What do you most often do with XML files?

   - Retrieve data from XML file.
   - Set or modify data of particular XML file.
   - Create new XML file(s).
   - Read and resend (web services).
   - Some other work: ……..……

5. Do you process large xml files?

   YES NO

Thank you for your time.

Appendix C

Interfaces of SDALX library

```
public interface IDocument : INode
{
        IDocumentType DocumentType { get; }
        DOMImplementation Implementation { get; }
        IElement DocumentElement { get; }

        IElement CreateElement(string tagName);
        IDocumentFragment CreateDocumentFragment();
        IText CreateTextNode(string data);
        IComment CreateComment(string data);
        ICDATASection CreateCDATASection(string data);
        IProcessingInstruction CreateProcessingInstruction(string
        target, string data);
        IAttribute CreateAttribute(string name);
        IEntityReference CreateEntityReference(string name);
        NodeList GetElementsByTagName(string tagname);
}
```

Figure 7.1: Interface for document node

```
public interface INode
{
        // node attributes
        string NodeName { get; }
        string NodeValue { get; set; }
        bool ReadOnly { get; }

        NodeType NodeType { get; }
        INode ParentNode { get; }
        NodeList ChildNodes { get; }
        INode FirstChild { get; }
        INode LastChild { get; }
        INode PreviousSibling { get; }
        INode NextSibling { get; }
        AttributeCollection Attributes { get; }
        IDocument OwnerDocument { get; }

        // Methods
        INode InsertBefore(INode newChild, INode refChild);
        INode ReplaceChild(INode newChild, INode oldChild);
        INode RemoveChild(INode oldChild);
        INode AppendChild(INode newChild);
        Boolean HasChildNodes();
        INode CloneNode(Boolean deep);
}
```

Figure 7.2: Interface for tree node

```
public interface IDocumentFragment : INode
{
}
```

Figure 7.3: Interface for Document fragment node

```
public interface IElement : INode
{
        string GetAttribute(string name);
        void SetAttribute(string name, string value);
        void RemoveAttribute(string name);
        IAttribute GetAttributeNode(string name);
        IAttribute SetAttributeNode(IAttribute newAttr);
        IAttribute RemoveAttributeNode(IAttribute oldAttr);
        NodeList GetElementsByTagName(string name);
        void Normalize();
}
```

Figure 7.4: Interface for Element node

```
public interface IAttribute : INode
{
        IElement OwnerElement { get; }
        bool HasDefaultValue { get; }
        Boolean Specified { get; }
}
```

Figure 7.5: Interface for Attribute node

```
public interface ICharacterData : INode
{
        string Data { get; set; }
        int Length { get; }

        string SubstringData(int offset, int count);
        void AppendData(string arg);
        void InsertData(int offset, string arg);
        void DeleteData(int offset, int count);
        void ReplaceData(int offset, int count, string arg);
}
```

Figure 7.6: Interface for Character data node

```
public interface IText : ICharacterData
{
	IText SplitText(int offset);
}
```

Figure 7.7: Interface for Text node

```
public interface ICDATASection : IText
{
}
```

Figure 7.8: Interface for CDATA node

```
public interface IComment : ICharacterData
{
}
```

Figure 7.9: Interface for Comment node

```
public interface IDocumentType : INode
{
	NamedNodeMap Entities { get; }
	NamedNodeMap Notations { get; }
}
```

Figure 7.10: Interface for Document type node

```
public interface IEntity : INode
{
	string PublicId { get; }
	string SystemId { get; }
	string NotationName { get; }
}
```

Figure 7.11: Interface for Entity node

```
public interface IEntityReference : INode
{
}
```

Figure 7.12: Interface for Entity reference node

```
public interface INotation : INode
{
        string PublicId { get; }
        string SystemId { get; }
}
```

Figure 7.13: Interface for Notation node

```
public interface IProcessingInstruction : INode
{
        string Target { get; }
        string Data { get; set; }
}
```

Figure 7.14: Interface for Processing instruction node

```
public interface IXmlDeclaration : INode
{
        string Encoding { get; set; }
        string Standalone { get; set; }
        string Version { get; }
}
```

Figure 7.15: Interface for Xml declaration node

Appendix D

Test results

Examples of test results for proposed SDALX API, for observation of all tests please take a look into attached DVD.

Q1:

```
Started parsing 100MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 5,06848 seconds.
Resolving query: 8,3048566 seconds.
Total time: 13,3733366 seconds.
Return nodes: 1
Number of fragments: 5
-------------------------------------------
Started parsing 200MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 15,1494016 seconds.
Resolving query: 14,4284386 seconds.
Total time: 29,5778402 seconds.
Return nodes: 1
Number of fragments: 12
-------------------------------------------
Started parsing 300MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 23,9903341 seconds.
Resolving query: 23,0315961 seconds.
Total time: 47,0219302 seconds.
Return nodes: 1
Number of fragments: 17
-------------------------------------------
Started parsing 400MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 31,1323284 seconds.
Resolving query: 29,6791961 seconds.
Total time: 60,8115245 seconds.
Return nodes: 1
Number of fragments: 23
-------------------------------------------
Started parsing 500MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 40,8519676 seconds.
```

```
Resolving query: 39,9524146 seconds.
Total time: 80,8043822 seconds.
Return nodes: 1
Number of fragments: 28
-----------------------------------------
Started parsing 600MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 36,1203275 seconds.
Resolving query: 44,9609533 seconds.
Total time: 81,0812808 seconds.
Return nodes: 1
Number of fragments: 34
-----------------------------------------
Started parsing 700MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 54,6158953 seconds.
Resolving query: 51,0826 seconds.
Total time: 105,6984953 seconds.
Return nodes: 1
Number of fragments: 39
-----------------------------------------
Started parsing 800MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 64,2839369 seconds.
Resolving query: 57,9965181 seconds.
Total time: 122,280455 seconds.
Return nodes: 1
Number of fragments: 45
-----------------------------------------
Started parsing 900MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 69,9005567 seconds.
Resolving query: 66,0617724 seconds.
Total time: 135,9623291 seconds.
Return nodes: 1
Number of fragments: 50
-----------------------------------------
Started parsing 1000MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 79,6915816 seconds.
Resolving query: 72,5813996 seconds.
Total time: 152,2729812 seconds.
Return nodes: 1
Number of fragments: 57
-----------------------------------------
```

Q2:

```
Started parsing 100MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 4,920319 seconds.
Resolving query: 7,9042041 seconds.
Total time: 12,8245231 seconds.
Return nodes: 1689
Number of fragments: 5
-----------------------------------------
Started parsing 200MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 9,7596948 seconds.
Resolving query: 15,9657645 seconds.
Total time: 25,7254593 seconds.
Return nodes: 3382
Number of fragments: 12
-----------------------------------------
Started parsing 300MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 23,9527233 seconds.
Resolving query: 23,6171782 seconds.
Total time: 47,5699015 seconds.
Return nodes: 5096
Number of fragments: 17
-----------------------------------------
Started parsing 400MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 29,4970125 seconds.
Resolving query: 32,6025708 seconds.
Total time: 62,0995833 seconds.
Return nodes: 6804
Number of fragments: 23
-----------------------------------------
Started parsing 500MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 41,9794807 seconds.
Resolving query: 41,6288848 seconds.
Total time: 83,6083655 seconds.
Return nodes: 8490
Number of fragments: 28
-----------------------------------------
Started parsing 600MB.xml document by SDALX.LargeXmlDocument class.
Splitting to small documents: 48,174878 seconds.
Resolving query: 47,7869727 seconds.
Total time: 95,9618507 seconds.
```

Return nodes: 10187

Number of fragments: 34

Started parsing 700MB.xml document by SDALX.LargeXmlDocument class.

Splitting to small documents: 49,8513668 seconds.

Resolving query: 55,8501746 seconds.

Total time: 105,7015414 seconds.

Return nodes: 11868

Number of fragments: 39

Started parsing 800MB.xml document by SDALX.LargeXmlDocument class.

Splitting to small documents: 68,326468 seconds.

Resolving query: 63,7249587 seconds.

Total time: 132,0514267 seconds.

Return nodes: 13593

Number of fragments: 45

Started parsing 900MB.xml document by SDALX.LargeXmlDocument class.

Splitting to small documents: 76,5357247 seconds.

Resolving query: 74,4190842 seconds.

Total time: 150,9548089 seconds.

Return nodes: 15345

Number of fragments: 50

Started parsing 1000MB.xml document by SDALX.LargeXmlDocument class.

Splitting to small documents: 272,8075969 seconds.

Resolving query: 91,9839528 seconds.

Total time: 364,7915497 seconds.

Return nodes: 17034

Number of fragments: 57

QU:

Started parsing uniref50.xml document by SDALX.LargeXmlDocument class.

Splitting to small documents: 1386,3980447 seconds.

Resolving query: 1750,6886087 seconds.

Total time: 3137,0866534 seconds.

Number of fragments: 869
